Heriot-Watt University

Integrating Distributed Data Streams

Alasdair John Graham Gray

October 7, 2007

Submitted for the degree of
Doctor of Philosophy in Computer Science
on completion of research in the
Department of Computer Science,
School of Mathematical and Computing Sciences.

Abstract

There is an increasing amount of information being made available as data streams, e.g. stock tickers, data from sensor networks, smart homes, monitoring data, etc. In many cases, this data is generated by distributed sources under the control of many different organisations. Users would like to seamlessly query such data without prior knowledge of where it is located or how it is published. This is similar to the problem of integrating data residing in multiple heterogeneous stored data sources. However, the techniques developed for stored data are not applicable due to the continuous and long-lived nature of queries over data streams.

This thesis proposes an architecture for a stream integration system. A key feature of the architecture is a republisher component that collects together distributed streams and makes the merged stream available for querying. A formal model for the system has been developed and is used to generate plans for executing continuous queries which exploit the redundancy introduced by the republishers. Additionally, due to the long-lived nature of continuous queries, mechanisms for maintaining the plans whenever there is a change in the set of data sources have been developed. A prototype of the system has been implemented and performance measures made.

The work of this thesis has been motivated by the problem of retrieving monitoring information about Grid resources. However, the techniques developed are general and can be applied wherever there is a need to publish and query distributed data involving data streams.

Dedication

To my parents: I wouldn't be the person that I am without them.

Acknowledgements

There are several people who have helped me out over the course of my PhD by offering me advice, support, or just a beer when it has been necessary. The following is by no means exhaustive, but are the people who must not be forgotten.

First, I would like to acknowledge all the help that I have received from my supervisors Werner Nutt and Howard Williams. They have both provided me with valuable input over the years which have helped me enormously. Additionally, the staff in the School of Mathematical and Computer Sciences have all been tremendously welcoming and supportive.

Second I would like to thank all of the R-GMA developers. They have been a valuable resource for exchanging ideas and providing technical support. I would also like to acknowledge the support and encouragement I have received from the database research community, particularly in the UK.

During my time at Heriot-Watt I have made many friends who have been there when I needed someone to talk ideas through with. In particular Fiona Bazoglu, Andy Cooke, Bill Ellis, Amanda Hearn, Lisha Ma, Manuel Maarek, Christine McBride, Kenneth McLeod, Claire Porter, Karen Sutherland, Liz Uruchurtu, and Mike Wicks.

I must not forget all my friends and flat buddies who have been there all the way through. Either for talking at me about their own thesis, Linda Franklin, and Kate Shires, or just to encourage and support me, Rohan Berry-Crickmar, Susie Brigg, Erica Darby, Pete Dunlop, Liz Green, and Niall McGregor.

I would not have been able to embark on an endeavour such as a PhD without the love and support of my family. My brother James was able to give advice as one who has been through it all. My parents, Alex and Cate, have always supported me in whatever I have done, but their patience and support have been invaluable. My sister Alison together with Sean and Charlie have been able to cheer me up. I would also like to thank Judy Abrams, and Mark and Diane Bromberg for all their encouragement.

Finally, it goes without saying that I would not have achieved this without the help, encouragement, friendship, and meals provided by my wife Katy, her Teddy and her cat Kipling. Katy, I love you with all my heart and thank you.

ACADEMIC REGISTRY
Research Thesis Submission

Name:	Alasdair J G Gray		
School/PGI:	MACS		
Version: *(i.e. First, Resubmission, Final)*	Final	Degree Sought:	Ph. D

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1) the thesis embodies the results of my own work and has been composed by myself
2) where appropriate, I have made acknowledgement of the work of others and have made reference to work carried out in collaboration with other persons
3) the thesis is the correct version of the thesis for submission*.
4) my thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying, subject to such conditions as the Librarian may require
5) I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.

* *Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is submitted.*

Signature of Candidate:	Alasdair Gray	Date:	12 October 2007

Submission

Submitted By *(name in capitals):*	C. PORTER
Signature of Individual Submitting:	
Date Submitted:	17/10/07

For Completion in Academic Registry

Received in the Academic Registry by *(name in capitals):*			
Method of Submission *(Handed in to Academic Registry; posted through internal/external mail):*	Handed into Academic Registry		
Signature:		Date:	17 Oct 07

Contents

1 Introduction **1**

1.1 Background 1

1.2 Summary of the Thesis 3

1.3 Key Contributions 4

1.4 Structure of Thesis 6

1.5 Publications 8

2 Background **11**

2.1 Data Integration 11

2.1.1 Relating Data Sources to the Global Schema 12

2.1.2 Semantic Integration 16

2.1.3 Query Execution 16

2.1.4 Existing Data Integration Systems 17

2.2 Publish/Subscribe Systems 18

2.2.1 Existing Publish/Subscribe Systems 20

2.3 Data Streams and Data Stream Processing 21

2.3.1 Data Stream Management Systems 22

2.3.2 Queries over Data Streams 23

2.3.3 Existing Data Stream Management Systems 25

2.3.4 Distributed Stream Processing 26

2.4 Summary 27

3 Motivation: Grid Information and Monitoring Systems **28**

3.1 Grid Computing 29

3.1.1 Computational Grids 29

3.1.2 Grid Projects 29
3.1.3 Components of a Grid 30
3.2 Requirements for a Grid Information and Monitoring System 31
3.2.1 Publishing Data 33
3.2.2 Locating and Querying Data 34
3.2.3 Scalability, Robustness, and Performance 34
3.2.4 Security 35
3.3 Existing Systems and Possible Approaches 35
3.3.1 Possible Approaches 35
3.3.2 Proposed Theoretical Architectures 37
3.3.3 Existing Systems 40
3.4 Summary 42

4 A Stream Integration System **43**
4.1 An Architecture for a Stream Integration System 43
4.1.1 A Virtual Dataspace 44
4.1.2 Roles and Agents 44
4.1.3 Global Schema 46
4.1.4 Producers and Consumers: Semantics 48
4.1.5 Republishers 50
4.1.6 The Registry 51
4.2 R-GMA: A Partial Implementation 52
4.2.1 The R-GMA Architecture 52
4.2.2 Query Answering in R-GMA 54
4.3 Comparison to Requirements and Existing Systems 56
4.3.1 Meeting the Requirements 56
4.3.2 Comparison to Other Grid Information and Monitoring Systems 57
4.4 Summary 57

5 Answering Continuous Queries Using Views **59**
5.1 A Formal Framework for Publishing and Querying Data Streams . . . 60
5.1.1 A Global Schema 60
5.1.2 Streams and Their Properties 60

5.1.3 Producing a Data Stream 63
5.1.4 Global Queries and Query Plans 65
5.2 Query Plans Using Republishers 66
5.2.1 Republishers and Queries over Republishers 66
5.2.2 Properties of Query Plans 67
5.2.3 Soundness 69
5.2.4 Completeness 69
5.2.5 Duplicate Freeness 71
5.2.6 Weak Order 72
5.3 Computing Consumer Query Plans 75
5.3.1 Relevant Publishers 76
5.3.2 Subsumption of Publishers 77
5.3.3 Plans Using Maximal Relevant Republishers 78
5.3.4 Discussion 82
5.4 Summary 83

6 Maintaining Query Plans **84**
6.1 Maintaining Consumer Query Plans 85
6.1.1 Adding a Producer 86
6.1.2 Deleting a Producer 87
6.1.3 Adding a Republisher 89
6.1.4 Deleting a Republisher 90
6.1.5 Discussion 94
6.2 Planning and Maintaining Republisher Queries 94
6.2.1 Requirements of a Publisher Hierarchy 95
6.2.2 Generating and Maintaining Republisher Query Plans 96
6.3 Summary 98

7 Implementation Details **100**
7.1 Implementing the Stream Integration System 101
7.2 The R-GMA System 103
7.2.1 Design of the Implemented R-GMA 104
7.2.2 Storing the Registration Data of the Components 106

7.2.3 Continuous Query Planning 110
7.2.4 Maintaining Continuous Query Plans 113
7.3 Improving the Query Planning and Plan Maintenance Mechanisms . . 115
7.3.1 Storing the Registration Data of the Components 116
7.3.2 Finding Relevant Publishers 120
7.3.3 Constructing the Query Plan 127
7.3.4 Improving the Plan Maintenance 131
7.4 Switching between Query Plans 135
7.5 Summary 138

8 Performance Measures **139**
8.1 Performance of the Registry Service 140
8.1.1 Experimental Method 141
8.1.2 Experimental Setup 142
8.1.3 Results 144
8.1.4 Discussion 162
8.2 Effects of a Publisher Hierarchy 165
8.2.1 Experimental Method 166
8.2.2 Experimental Setup 166
8.2.3 Results 167
8.2.4 Discussion 173
8.3 Summary 174

9 Conclusions **176**
9.1 Review of Thesis and Conclusions 176
9.2 Overcoming Incomplete Data 179
9.3 Future Work 181
9.3.1 Generating Query Plans 182
9.3.2 Protocols for Plan Switching 182
9.3.3 Increased Query Functionality 183
9.3.4 Technology Uptake 184

List of Tables

7.1 An instance of the database used by the registry service in R-GMA. . 109

7.2 The base values and their interpretation for the flags attribute. 110

7.3 The result of executing Query (7.11) to the registry database instance in Table 7.1. 113

7.4 The result of executing Query (7.13) to the registry database instance in Table 7.1. 115

7.5 An instance of the database used by the improved registry service. . . 119

7.6 Abbreviations used in the presentation of the improved registry database instance. 120

7.7 Boundary and value criteria for when an interval a, b contradicts an interval c, d. 122

7.8 The result of executing Query (7.22) to the registry database instance in Table 7.5. 124

7.9 The result of executing Query (7.27) to the registry database instance in Table 7.5. 134

8.1 The mean and variance for 50 relevant consumers. 147

8.2 The mean and variance for 50 non-relevant consumers. 151

8.3 The mean and variance for 50 non-relevant consumers run in reverse. 154

8.4 The mean and variance for 50 consumers with a mixture of queries. . 159

8.5 The mean and variance for 100 relevant consumers. 162

8.6 Average time and variance to deliver a tuple using different numbers of republishers. 168

8.7 Peak values for the republishers. 169

8.8 Average time and variance to deliver a tuple using different numbers of republishers when the producer introduces no delay between inserting each tuple. 170
8.9 Average time to deliver a tuple using different numbers of republishers; no delays are introduced by the republishers between each poll of their input queues. 172

List of Figures

2.1 Generic architecture of a data integration system. 12

2.2 Matching tree for two stock subscriptions. 19

2.3 Abstract architecture for a Data Stream Management System. 23

3.1 The major components of DataGrid/gLite middleware. 32

3.2 The components of the GMA and their interactions. 38

3.3 The components of the Generic Monitoring and Information System Model and their interactions. 39

4.1 The roles and agents of the stream integration system design along with their interactions. 45

5.1 Publisher configuration $\mathcal{P}_0$ with the plans derived for queries q_1 and q_2. 81

6.1 Publisher configuration $\mathcal{P}_0$ with the plans derived for queries q_1 and q_2. 86

6.2 The data connections of Publisher Configuration $\mathcal{P}_1$ and the effects on queries q_1 and q_2. 88

6.3 Consumer queries q_1 and q_2 being posed at publisher configuration $\mathcal{P}_2$. 91

6.4 Consumer queries q_1 and q_2 being posed at publisher configuration $\mathcal{P}_3$. 93

6.5 The result of using consumer planning techniques for republisher queries. 95

7.1 A UML component diagram of the services of the R-GMA system. . . 105

7.2 R-GMA registry database schema. 107

7.3 Improved registry database schema. 118

7.4 Algorithm to generate equivalence classes of republishers. 129

8.1 Registry service performance with 50 relevant consumers (1). 145

8.2 Registry service performance with 50 relevant consumers (2). 146

8.3 Results from registry service performance test with 50 non-relevant consumers (1). 149
8.4 Results from registry service performance test with 50 non-relevant Consumers (2). 150
8.5 Results from registry service performance test with 50 non-relevant consumers, producers added in reverse (1). 152
8.6 Results from registry service performance test with 50 non-relevant consumers, producers added in reverse (2). 153
8.7 Graphs showing the average time taken to add producers when there are 50 non-relevant consumers. 155
8.8 Results from registry service performance test with 50 mixed consumers (1). 157
8.9 Results from registry service performance test with 50 mixed consumers (2). 158
8.10 Results from registry service performance test with 100 relevant consumers (1). 160
8.11 Results from registry service performance test with 100 relevant consumers (2). 161
8.12 Graphs showing the average time taken to add a producer as the number of conditions in the queries increases. 163
8.13 Time taken to deliver tuples from the producer to the consumer using different numbers of republishers. 167
8.14 Time taken to deliver tuples from the producer to the consumer using different numbers of republishers when the producer introduces no delay between inserting each tuple. 171
8.15 Time taken to deliver tuples from the producer to the consumer using different numbers of republishers which introduce no delay between polls of their input queues. 172

Chapter 1

Introduction

1.1 Background

Streams of data are increasingly becoming available from multiple, autonomous, distributed sources. Typical examples of such data streams are stock tickers [1,2], sensors and sensor networks [3,4], and various monitoring data including environmental [5,6], traffic [7], network [8,9], and computing resources [10,11]. Such streams originate from a variety of sources including "dumb" sensors, which simply make readings about their environment available, to sophisticated computers publishing their status information and which have the ability to perform operations on their data along with data collected from other sources.

The applications of streaming data are many and varied, from monitoring the situation in a disaster, e.g. a fire in the Underground, to scheduling and tracking jobs on a computational Grid [12]. The users of the data are typically distributed and have different requirements on the information they retrieve. For example, consider the situation of a computational Grid where computational, network, and storage resources are offered by various research groups which are located around the world. To allow jobs to be scheduled, a resource broker needs the *latest* information about the status of the resources on the Grid along with their capabilities. A user who has issued a job will want to *continuously* track the progress of their job using a visualisation tool. A resource manager will be interested in how much their resource is being utilised which requires that *histories* of the monitoring streams are kept and are available for querying.

A variety of approaches have been developed for querying streams based on the requirements of the users of such data. The simplest set of systems which focus on delivering data based on its content are publish/subscribe systems [13, 14]. These allow for the publication of data as discrete events. Users declare an interest in events with certain characteristics. When an event is published it may match multiple subscriptions and would be forwarded on to each.

Demand for more sophisticated techniques for filtering, manipulating and relating stream data has resulted in intense research in data streams over the last 10 years [15–17]. A variety of stream processing systems have been developed that are able to manipulate, store, and query multiple streams of data. However, the majority of these systems are based on a centralised approach using a single server. They require all the data to flow to some central point where it is processed before the answers to queries are streamed to the users. Some distributed stream processing engines are now starting to emerge [18–20], although these are akin to a distributed database management system [21] in that all of the data and processing resources are under the control of one organisation.

Increasingly, users are wanting to relate data in streams being published by multiple, autonomous, distributed sources without needing to locate, retrieve, and process the streams themselves. One example would be a Grid scheduler [22] which must use streams of monitoring data about resources on a Grid that are provided by multiple organisations. The requirement to be able to combine and manipulate data from multiple distributed autonomous streaming data sources without needing to know specific details of any of the data sources is similar to the idea behind data integration.

Data integration allows heterogeneous stored data sources, e.g. databases, web pages, etc., to be accessed through a common schema as if they were a *virtual database* [23–26]. The data resides at the sources, each of which has its own schema to describe its data. The sources can be queried using the common schema which allows the data residing in the multiple sources to be retrieved and combined without the user needing to know what data sources exist or how to relate and convert the source data. However, the techniques are not directly applicable to distributed data streams due to the difference in nature of a query involving a data stream as opposed to a query over a database.

Querying streams of data is based on a different paradigm to that of querying a database. A database management system is designed to query a specific instance of a stored data set whereas a stream query is generally interested in the changing values on the stream. Specifically, when a query is executed by a database management system the data is "frozen" for the duration of the query. In other words, the query is executed against a *static* instance of the data that currently resides in the database and returns those tuples in the database instance that satisfy the condition of the query. On the other hand, the most common type of stream query, known as a *continuous query* [27], is executed against data in a stream as it arrives on the stream and returns those tuples in the stream that satisfy the query condition. A continuous query is long-lived and returns those newly published tuples that answer the query whereas a database query is executed once and returns only those tuples that exist in the database at that moment in time.

So far, there has been no work considering how the ideas and techniques of data integration could be applied and extended to meet the demands of distributed stream processing. This thesis will consider how to integrate distributed data, both stored and streaming, published by autonomous, distributed sources to give the user the appearance of a *virtual global data space*. The work will be motivated by the requirements of a Grid information and monitoring system which must be able to query streams of data, along with stored data, about the resources on a computational Grid that are provided by multiple organisations. Although the work will be motivated by the requirements of a Grid information and monitoring system, the techniques developed are general and could be applied wherever there is a need to publish and query distributed data sources publishing both stored and streaming data.

1.2 Summary of the Thesis

The aim of this thesis is to develop mechanisms by which autonomous distributed data sources, both streaming and stored, can be queried in a unified and efficient manner without the user needing to know the existence or location of individual data sources.

This will be achieved through the following objectives.

Analysis of existing approaches: An analysis of data integration techniques along

with data stream processing methods will be conducted. The results of this analysis will be used to inform the design and prototype implementation of a stream integration system.

Design of a stream integration system: A system that is capable of publishing and querying distributed autonomous data sources, both streaming and stored, will be designed. The system should present to the user a query interface that presents a virtual dataspace that consists of all the streams and stored data made available by the distributed data sources. The system needs to provide a mechanism to efficiently answer multiple, simultaneous queries by materialising partial answers.

Framework for integrating streams: A formal model for a data stream will be developed that allows desirable properties of streams to be defined. The model should capture the publication and querying of distributed streams. Techniques for generating plans to answer a continuous query will be developed and shown to return the correct answer and that the answer will guarantee desirable properties, e.g. being duplicate free.

Mechanisms to adapt to changes: Queries over data streams are long-lived. During the execution of a query, the set of available streams will change. The system must be able to adapt to the change so that the answers returned to a query continue to be correct.

Implementation of a prototype: A prototype of part of the designed system will be implemented. Suitable existing platforms will be considered and built upon. Performance tests will be conducted to compare some of the design choices and to investigate the efficiency of the resulting planning mechanism.

1.3 Key Contributions

This thesis makes the following contributions.

- An architecture for integrating autonomous distributed data sources, both streaming and stored, through a *virtual dataspace*. The architecture allows both

streaming and stored data sources, each with their own schema, to have their data accessed by queries over a single global schema.

The architecture is based on a logical model for data streams which allows key properties of a stream to be defined, e.g. when a stream is duplicate free. The architecture uses the data stream model to formalise the publication of a data stream and provides semantics for queries over the global schema and over the schemas of the data sources. As part of these semantics, key properties that an answer stream to a continuous query over the integrated global schema should possess are identified and defined.

A key feature of this architecture is the *republisher* component which poses a query over the global schema and makes the resulting answer available for other queries. The republisher allows queries to be answered more efficiently by collecting and merging the data from several sources together and making the resulting data available from a single point.

- Techniques that allow a continuous query over the global schema to be translated into suitable queries over the data sources. This is not straightforward since the semantics of continuous queries have to be taken into account to ensure that the answer streams returned have desirable and well defined properties e.g. containing only tuples that answer the query and containing all such tuples.

  In particular, the techniques developed generate query plans for answering continuous global queries in the presence of republishers. The republishers complicate query answering as they introduce redundancy into the data which means a choice has to be made as to where to retrieve each stream. The approach developed ensures that the answer stream to a query possesses desirable properties whilst minimising the number of publishers, i.e. stream sources and republishers, that are contacted to retrieve the data.

- A mechanism to maintain the query plans when the set of available publishers changes. A continuous query is long-lived and as such, during the execution of the query the set of publishers can change with publishers being added or removed. These changes can affect the answer stream generated by a query plan for a continuous query. For example, if a republisher is being used to answer a

query and that republisher is removed from the system, then the answer stream will stop even though the data, which is published by a data source, is still flowing in the system. The mechanism developed identifies the queries that are potentially affected by a change in the set of publishers, and when a plan is affected the mechanism will update the plan to reflect the new set of publishers.

- A prototype implementation of the query planning and maintenance mechanisms required for a stream integration system. The prototype implements republishers so that they can form a hierarchy of publishers, i.e. a republisher may use the answer stream computed by another republisher to answer its query. The prototype is used to investigate the efficiency of query answering, and the effects on query response times introduced by using a hierarchy of publishers.

1.4 Structure of Thesis

The thesis has been organised into 9 chapters as follows.

Chapter 2 presents a discussion of the key issues in the fields of research relevant to this thesis. It begins by giving an overview of data integration with the general concepts and details of the techniques developed for presenting a virtual database to the user, i.e. how to relate the schema of the data sources to some common schema. Some key integration systems are highlighted.

The chapter then goes on to discuss techniques for handling streams of data. The concepts of publish/subscribe systems are briefly introduced along with details of some key systems. Then more expressive stream processing techniques are presented. An overview of a data model for streams is presented along with a discussion about the semantics of stream queries. Finally, details of key centralised and distributed data stream processing systems are presented.

Chapter 3 presents the motivation behind the work in this thesis. The motivating problem for this thesis is that of providing information and monitoring data about resources on a Grid. It will be shown that the idea of integrating streams of data is well suited to the motivating problem.

The chapter begins with a brief introduction to the concepts and ideas behind Grid computing. This allows the requirements for a Grid information and monitoring system to be identified. A discussion of the existing approaches to providing information and monitoring data about resources on a Grid is presented with two theoretical models and several existing systems analysed.

Chapter 4 presents an architecture for integrating distributed data sources, including both streaming and stored data sources. The components of the architecture along with their roles and interaction within the architecture are presented.

One of the key features of the architecture is its use of *republisher* components which collect and merge data from several data sources making the resulting data available from a single location. This allows the architecture to scale to large numbers of data sources and queries, as partially computed results at the republishers can be used to answer more general queries.

The chapter concludes with a discussion of the R-GMA system which partially implements the architecture presented. R-GMA has been developed as a Grid information and monitoring system and its functionality will be compared with that of other existing systems.

The next two chapters give details of the theoretical framework and the mechanisms developed within that framework for answering continuous queries efficiently.

Chapter 5 presents the formal framework for integrating streams of data. It begins by presenting a formal model for a data stream along with defining desirable properties of a stream.

The model is then used to present a formalism for publishing and querying autonomous distributed data streams according to some common schema. The republisher components of the framework, while providing a scalable and efficient way to answer queries, complicate the process of generating a plan to answer the query. This is because they introduce a choice in where to retrieve each part of the answer stream to a query. To ensure that a query answer is correct, properties that a query plan should guarantee are identified and defined. Finally, an approach to generating correct query plans based on identifying and ranking relevant publishers is presented.

Chapter 6 introduces the problem of maintaining query plans when there is a change in the set of publishers. Since queries are long-lived, any such change could have an effect on the answer streams generated for existing plans. The mechanisms developed identify those query plans that are affected and how to update the plan without needing to re-plan the whole query.

The chapter then goes on to consider how the query planning and maintenance mechanisms developed can be applied to republishers. Since republishers can consume streams from other republishers, a hierarchy of publishers results. The query planning and maintenance mechanisms should ensure that the resulting hierarchies have desirable properties, e.g. do not contain a cycle where a republisher R_1 consumes from another republisher R_2 and R_2 consumes from R_1.

Chapter 7 details the implementation of a prototype to show that the algorithms and mechanisms developed in the previous two chapters work in practice. It is shown that the existing R-GMA system provides a good framework within which to implement the query planning and maintenance algorithms developed. As such, details of the existing R-GMA implementation along with its query planning and plan maintenance techniques are presented.

Chapter 8 presents the experiments and results conducted to collect performance measures from the prototype. The first of the experiments investigates the performance of the query planning mechanism and is used to guide the development of the prototype. The second experiment investigates the effects on the latency of an answer tuple of using a hierarchy of publishers.

Finally, Chapter 9 presents the conclusions of this thesis, and suggests further work to extend the results. The extensions suggested would allow the stream integration system to be applicable in a larger range of applications.

1.5 Publications

The work of this thesis has been reported in the following papers.

- A. Cooke, A.J.G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A. Wilson, R. Cordenonsi,

L. Cornwall, A. Djaoui, S. Fisher, N. Podhorszki, B. Coghlan, S. Kenny, and D. O'Callaghan. R-GMA: An information integration system for grid monitoring. In *Proceedings of On the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE—OTM Confederated International Conferences*, volume 2888 of *Lecture Notes in Computer Science*, pages 462–481, Catania (Italy), November 2003. Springer-Verlag.

- A.W. Cooke, A.J.G. Gray, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, L. Field, S.M. Fisher, S. Hicks, J. Leake, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny, D. O'Callaghan, and J. Ryan. The relational grid monitoring architecture: Mediating information about the grid. *Journal of Grid Computing*, 2(4):323–339, December 2004.

- A. Cooke, A.J.G. Gray, and W. Nutt. Stream integration techniques for grid monitoring. *Journal on Data Semantics*, 2:136–175, 2005.

- A.J.G. Gray and W. Nutt. Republishers in a publish/subscribe architecture for data streams. In *Proceedings of 22nd British National Conference on Databases (BNCOD22)*, volume 3567 of *Lecture Notes in Computer Science*, pages 179–184, Sunderland (UK), July 2005. Springer-Verlag.

- A.J.G. Gray and W. Nutt. A data stream publish/subscribe architecture with self-adapting queries. In *Proceedings of On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE—OTM Confederated International Conferences*, volume 3760 of *Lecture Notes in Computer Science*, pages 420–438, Agia Napa (Cyprus), October 2005. Springer-Verlag.

In addition to the work in this thesis, the topic of incompleteness in data streams has been investigated. The key results of the work on incompleteness are briefly described in Section 9.2 and have been reported in the following papers.

- A.J.G. Gray, W. Nutt, and M. Howard Williams. Sources of incompleteness in grid publishing. In *Proceedings of 23rd British National Conference on Databases (BNCOD23)*, volume 4042 of *Lecture Notes in Computer Science*, pages 94–101, Belfast (UK), July 2006. Springer-Verlag.

- A.J.G. Gray, M.H. Williams, and W. Nutt. Answering arbitrary conjunctive queries over incomplete data stream histories. In *Proceedings of 8th International Conference on Information Integration and Web-based applications and Services (iiWAS2006)*, pages 259–268, Yogyakarta (Indonesia), December 2006. Austrian Computer Society.

- A.J.G. Gray, W. Nutt, and M.H. Williams. Answering queries over incomplete data stream histories. *International Journal of Web Information Systems*, 2007. Invited submission that has been accepted to appear subject to minor revisions.

Chapter 2

Background

In this chapter the research topics that are relevant to this thesis will be introduced with important results in the literature presented. The work of this thesis touches on the areas of data integration, publish/subscribe systems, and data stream processing.

2.1 Data Integration

Data integration aims to allow a collection of heterogeneous distributed data sources, e.g. a collection of e-commerce websites or a collection of relational databases, to be queried as if they were a single homogeneous *virtual database*. To enable a collection of data sources to appear as a virtual database requires a system to present a schema for a user to query. The data integration system then transforms the query over the schema into one or more queries against the available data sources in order to answer the user query [25, 26, 28].

Wiederhold envisioned an architecture for data integration where collections of sources are queried via a global schema as opposed to the individual source schemas [23], see Figure 2.1 which has been reproduced from [29]. The architecture involves the data sources being exposed as if they are a relational database through a *wrapper* which is responsible for querying the local data source. A *mediator* component is responsible for receiving a user query over the global schema and transforming it into one or more sub-queries to pose to the wrappers. The final answer to the user query is achieved by generating an execution plan which retrieves the data from the sources and uses the mediator or a wrapper to combine the answer sets from the sub-queries,

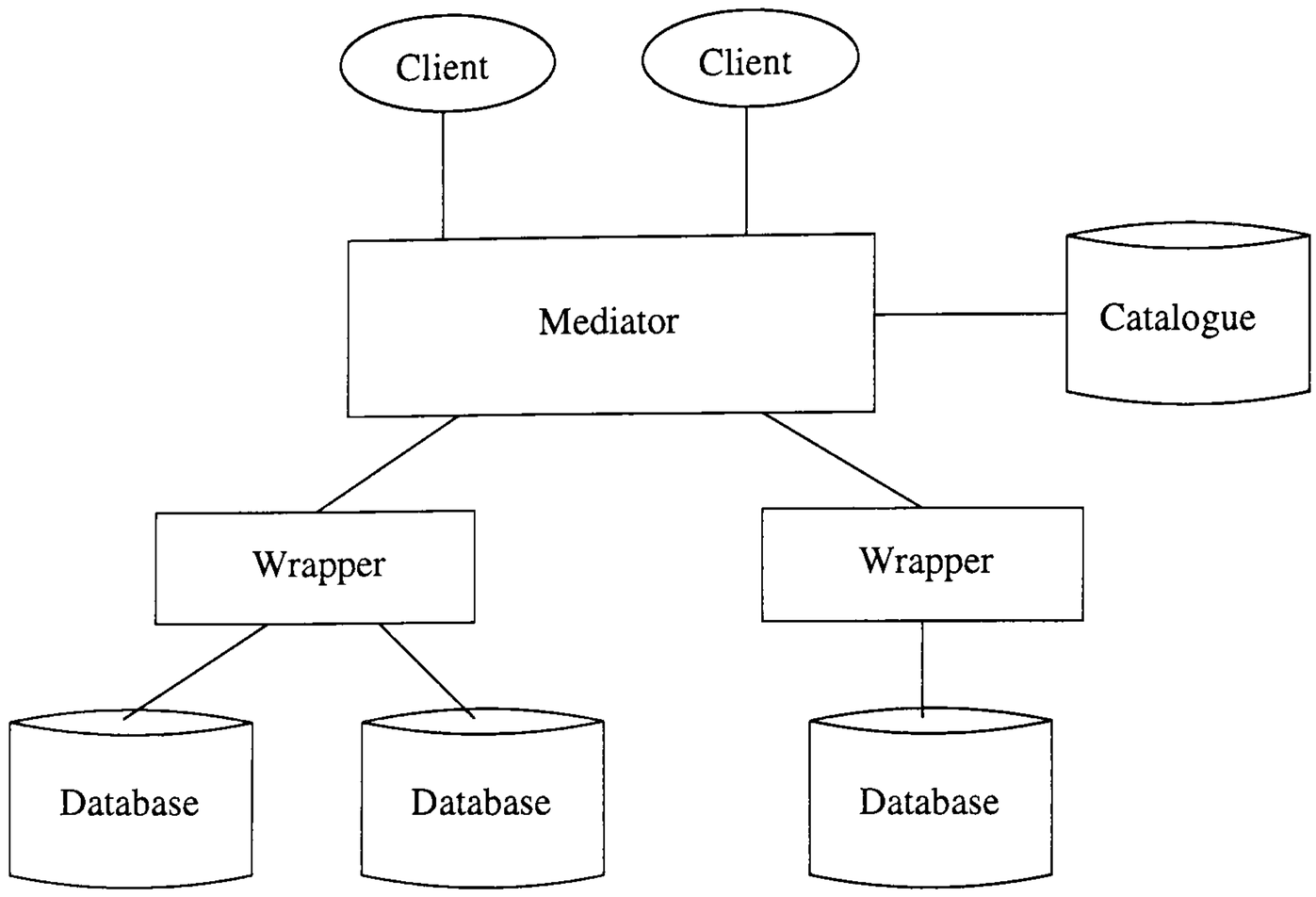


Figure 2.1: Generic architecture of a data integration system.

e.g. for the execution of a join involving data from multiple data sources. The schema of the virtual database is stored in the catalogue along with a mapping relating each source with the schema of the virtual database.

2.1.1 Relating Data Sources to the Global Schema

As shown in Figure 2.1 the virtual database is made up of a collection of sources each appearing as a separate relational database. Each source is managed independently, and thus has its own *local schema* to which the data conforms. The sources are integrated into the virtual database by means of a *global schema*. Ullman, in his paper [24], identified two types of schema level integration which were named by Levy [30] as "global as view", often referred to as GAV, and "local as view", often referred to as LAV. Each of these will now be addressed in turn.

Global as View

In the global as view approach, the data made available by the data sources is described as a view over the data sources. That is, each global relation is defined as a query involving the data sources.

As an example of the global as view approach consider the global schema

$$\mathsf{Person}(\mathsf{name},\ \mathsf{address},\ \mathsf{age},\ \mathsf{job}), \tag{2.1}$$

to represent information about a person, storing their name, address, age and job title. Now consider that there are two data sources in the system:

1. A company database storing information about employees. This contains the relations

$$\mathsf{Employee}(\mathsf{NI\#},\ \mathsf{name},\ \mathsf{age},\ \mathsf{job}) \tag{2.2}$$

$$\mathsf{Address}(\mathsf{NI\#},\ \mathsf{address}), \tag{2.3}$$

   where Relation (2.2) stores the name, age, and job title of an employee using their national insurance number as a key. Relation (2.3) stores addresses, relating each address to an employee based on the national insurance number.

2. A university's database storing information about the students, containing the relation

$$\mathsf{Student}(\mathsf{matric\#},\ \mathsf{name},\ \mathsf{address},\ \mathsf{age}), \tag{2.4}$$

   which stores the name, address, and age of the students based on their matriculation number.

The Datalog [31] mapping in the catalogue, relating the local relations to the global schema, would be

$$\mathsf{Person}(\mathit{name},\ \mathit{address},\ \mathit{age},\ \mathit{job}) \leftarrow \mathsf{Employee}(\mathit{NI\#},\ \mathit{name},\ \mathit{age},\ \mathit{job})\ \&\ \mathsf{Address}(\mathit{NI\#},\ \mathit{address}) \tag{2.5}$$

$$\mathsf{Person}(\mathit{name},\ \mathit{address},\ \mathit{age},\ \text{'student'}) \leftarrow \mathsf{Student}(\_,\ \mathit{name},\ \mathit{address},\ \mathit{age}). \tag{2.6}$$

View (2.5) provides a tuple to the global relation Person when there is a tuple in Employee and a tuple in Address which contain the same value for the attribute $NI\#$. View (2.6) provides a tuple to the global relation Person when there is a tuple in Student. The "_" is interpreted as a variable that appears only once and whose value is not output.

Query planning in the global as view approach is reasonably straightforward. The query posed over the global schema is transformed into queries over the data sources by substituting the view definitions for each occurrence of a predicate. For example, consider the query

$$\mathsf{q}(\mathit{address}) \leftarrow \mathsf{Person}(\text{'john smith'},\ \mathit{address},\ \mathit{age},\ \mathit{job}), \tag{2.7}$$

which is asking for the address of all the people called 'john smith'. By replacing the global relation occurring in the query with the view definitions (2.5) and (2.6) the following queries are generated which the mediator must pose against the sources

$$\mathsf{q}_1(address) \leftarrow \mathsf{Employee}(NI\#,\ \text{'john smith'},\ age,\ job)\ \&$$
$$\mathsf{Address}(NI\#,\ address) \tag{2.8}$$

$$\mathsf{q}_2(address) \leftarrow \mathsf{Student}(matric\#,\ \text{'john smith'},\ address,\ age). \tag{2.9}$$

The execution of these queries against the data sources must be co-ordinated by the mediator. Techniques for co-ordinating the execution of these queries against distributed sources are discussed in Section 2.1.3.

The drawback of the global as view approach is that adding a new data source often requires the global schema to change. This means that to add a new data source, all of the mappings for the existing sources must be altered.

Local as View

In the local as view approach the data made available by each data source is described as a view over the global schema. That is, each relation made available by a data source is described by a query over the global schema.

As an example of the local as view approach consider the global schema

$$\mathsf{Address}(\mathsf{name},\ \mathsf{street},\ \mathsf{city}) \tag{2.10}$$

$$\mathsf{Phone}(\mathsf{name},\ \mathsf{number}). \tag{2.11}$$

Relation (2.10) gives the name and address of someone, and Relation (2.11) gives the name and phone number. Say there are two data sources in the system:

1. A phone book with the local schema

$$\mathsf{PhoneBook}(\mathsf{name},\ \mathsf{city},\ \mathsf{number}), \tag{2.12}$$

which provides the name, city, and telephone number of someone.

2. An address book with the local schema

$$\mathsf{AddressBook}(\mathsf{name},\ \mathsf{street},\ \mathsf{city}), \tag{2.13}$$

which stores the name, the street that they live on, and the city of someone.

The mapping stored in the catalogue using Datalog notation which relates the global relations to the available data sources would be

$$\mathsf{PhoneBook}(name,\ city,\ number) \leftarrow \mathsf{Address}(name,\ street,\ city)\ \&$$
$$\mathsf{Phone}(name, number), \tag{2.14}$$
$$\mathsf{AddressBook}(name,\ street,\ city) \leftarrow \mathsf{Address}(name,\ street,\ city). \tag{2.15}$$

The view definitions give the properties that tuples must have but they do not guarantee to provide all tuples that conform to the definition. There is also no guarantee of consistency between the sources.

A query that may be posed against this system could be

$$\mathsf{q}(city) \leftarrow \mathsf{Address}(\text{'john smith'},\ street,\ city), \tag{2.16}$$

which would be locating the city of anyone called 'john smith'. It is not immediately clear how to generate an answer to this query as the tuples are stored in the data sources which are related to the global schema by the view definitions (2.14) and (2.15). Several algorithms [32–37] have been developed to generate answers to a global query. These involve generating a rewriting of the query over the global schema into queries over the available local schemas. For the example query (2.16), the rewriting approach would generate the following queries over the data sources

$$\mathsf{r}_1(city) \leftarrow \mathsf{PhoneBook}(\text{'john smith'},\ city,\ \_) \tag{2.17}$$
$$\mathsf{r}_2(city) \leftarrow \mathsf{AddressBook}(\text{'john smith'},\ \_,\ city). \tag{2.18}$$

To retrieve the fullest set of answers to the query, both rewritings would be used to retrieve the data. However, this will lead to duplicate answers if the data sources contain the same information. In fact the situation is worse than this, if two instances of the same city are returned then these cannot be distinguished between being two different 'john smith' in the same city and the same instance being duplicated by two data sources [38].

The advantages of the local as view approach are twofold. The first is that data sources can be added, or removed, from the data integration system without affecting the views of any other source or the global schema. The second is that the content of the sources may be described more accurately as conditions can be placed on the

attributes. For example, say that the AddressBook in (2.15) only contained addresses in Edinburgh then the additional constraint

$$city = 'Edinburgh', \tag{2.19}$$

could be added to the view description.

2.1.2 Semantic Integration

The wrapper component has an important semantic role not mentioned thus far in the discussion about local as view and global as view mappings. Consider again the schemas for Person in the global schema (2.1) and Employee in the local schema (2.2). Previously, it was implicitly assumed that the age attribute in both schemas used the same units. However, it is feasible that the data in the source is actually stored as a date of birth. The wrapper constructed for the source would need to be able to convert the data in the source to the required format. For the age example here this is straightforward, but other cases are a lot more complex. For example, involving splitting attributes, problems of granularity of measurements or scales that are not easily converted, e.g. qualitative scales, [39, 40]. This problem is not unique to data integration and occurs in federated systems [41] and data warehousing [42]. As a consequence, it has been a topic of intense research for a number of years [43–45].

2.1.3 Query Execution

The techniques for the organisation of the execution of a query plan over a set of distributed data sources is the same for both the global as view and the local as view approaches. An overview of techniques for distributed query execution are presented in [29].

The techniques are based on the standard architecture for a data integration system (Figure 2.1). The wrappers inform the mediator of the querying capabilities of their source, and possibly cost estimates for executing queries. The mediator, based on this information, can then pass the wrappers suitable sub-queries in order to execute the query plan as efficiently as possible. The mediator may have additional cost models, along with the capability to perform certain operations such as joins across sources, to aid the efficient execution of queries.

Other relevant work in this area includes adaptive query processing and distributed query processing on the Grid (See Section 3.1 for more details about Grid computing). The idea behind adaptive query processing is to adjust the execution of queries based on the state of the system. Work has been conducted in this field since the first relational query processors [46, 47], which kept statistics about the size of relations. These techniques have been refined and developed in order to be applied in a distributed setting. An overview can be found in [48].

The OGSA-DAI project[1] [49] has developed services to publish and query data on a Grid. The querying service, OGSA-DQP [2] [50], plans the execution of a query, that may involve several data sources, by exploiting query execution services and parallel query processing techniques.

2.1.4 Existing Data Integration Systems

This section gives a brief overview of some key data integration systems.

Infomaster: [3] follows a local as view approach to data integration [36, 51]. The mediator component uses the inverse rules algorithm [36] which applies computational logic techniques.

INFOMIX: [4] follows a global as view approach focusing on using integrity constraints to combine data from inconsistent and incomplete data sources [52]. This allows it to reduce the number of sub-queries as some would never yield an answer due to violating the integrity constraints.

Information Manifold: follows a local as view approach to data integration [33]. The mediator applies techniques for answering queries using views giving rise to the bucket algorithm.

TSIMMIS: [5] follows a global as view approach to data integration [53]. The mediator component applies substitution techniques to translate the query over the global schema into a collection of queries over the local schemas.

---

[1] `http://www.ogsadai.org.uk/` (June 2007)

[2] `http://www.ogsadai.org.uk/about/ogsa-dqp/` (June 2007)

[3] `http://infomaster.stanford.edu/infomaster-info.html` (January 2007)

[4] `http://sv.mat.unical.it/infomix/` (January 2007)

[5] `http://infolab.stanford.edu/tsimmis/` (January 2007)

2.2 Publish/Subscribe Systems

The aim of a *publish/subscribe* system is to provide a flexible, dynamic, loosely coupled, scalable mechanism for distributed message delivery. The system consists of *producers* of information (which publish events into the system) and *consumers* of information (which subscribe to certain information). The system is responsible for the delivery of events from the producers to the relevant consumers without either being aware of the specific details of the other [13, 14].

There are two varieties of publish/subscribe systems, subject-based and content-based, which are distinguished by how events are matched to subscriptions. Early publish/subscribe systems were all of the first type, *subject-based*[6]. In subject-based systems messages are tagged with a topic to which they conform. The topics available are defined by the system and can be arranged into a hierarchy. The matching of subscriptions to messages is relatively straightforward as the subscriber declares what subjects they are interested in. The subjects each have a unique key and this can be quickly matched. For example, consider a system for publishing the current price of stock. The subjects available in the system could be different types of company, e.g. banks, electricity suppliers, etc. A subscriber interested in the price of electricity suppliers would subscribe to the electricity topic.

The second variety are *content-based* systems where subscribers register a pattern or query. For example, in a stock situation a subscriber could be interested in all electricity stock that has a price of less than 200 pence. This would be expressed as

$$\{\texttt{type} = \texttt{electricity},\ \texttt{price} \leq 200\}. \tag{2.20}$$

The advantage of the content-based systems is that the subscribers do not need to receive messages that they are not really interested in. However, the task of matching messages to subscriptions is more computationally demanding.

The task of matching events to content-based subscriptions is performed by a *mediator* sometimes called a broker. The mediator must be able to filter messages according to the subscriber's subscription but in a scalable manner. The majority of systems achieve this through a *matching tree algorithm* [54]. In a matching tree algorithm, the subscriptions are preprocessed so that the commonality between two

[6] Also referred to as topic-based.

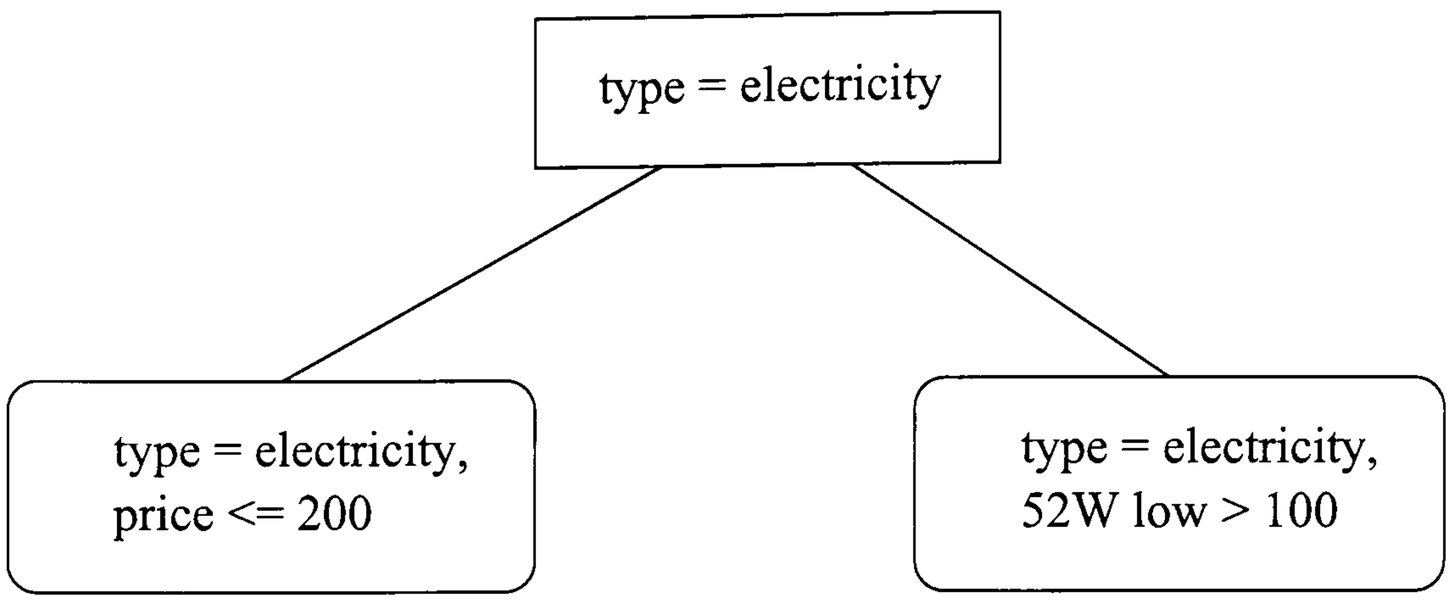


Figure 2.2: Matching tree for two stock subscriptions.

subscriptions can be exploited. This results in the leaves of the tree containing subscriptions but the nodes contain filter items. For example, consider the subscription in Equation (2.20) and the subscription

$$\{\texttt{type} = \texttt{electricity},\ \texttt{52 Week low} > 100\}. \tag{2.21}$$

These would form the matching tree shown in Figure 2.2. The matching of events to subscriptions must be conducted for each event published into the system.

There have been a variety of architectures used to implement publish/subscribe systems. These can be classified as either client-server or peer-to-peer. Within the client-server model, the publishers and subscribers are the clients and there are servers which are responsible for receiving, possibly storing, and forwarding the events. Several different topologies have been adopted: star topology with one central server, hierarchical topology with servers organised into a hierarchy, or irregular polygon topology with different protocols for server-server connections and server-client connections.

In the peer-to-peer model, each peer takes on part of the responsibility of the server in the client-server model. Again, different topologies of peer-to-peer networks have been used. For example, every peer being equal with connections to some, possibly all, other peers or super peer networks where each peer is connected to a super peer and the super peers are interconnected in some way.

An area which still requires additional research is that of security. Many of the systems require publishers and subscribers to authenticate themselves to the system in order to be able to publish or retrieve data. There have also been mechanisms developed to limit the events that an individual client may publish or subscribe to.

Thus limiting certain events to certain groups of users. However, content-based systems publish their information as plain text in order for the matching algorithms to function. Some preliminary work on encrypting messages has been presented in [55]. although due to the repetition in the content of messages, the encryption mechanism can be broken more easily.

Publish/subscribe has been touted as a generic mechanism for passing events around in a distributed manner. However, there has yet to be an application identified that fits the generic publish/subscribe paradigm. An analysis of the applications suggested for publish/subscribe, which includes video on demand, software updates, travel applications, and on-line gaming, show that each has different demands on a messaging system and thus no single standard has so far been agreed [56].

2.2.1 Existing Publish/Subscribe Systems

This section presents several publish/subscribe systems detailing whether they are subject or content based, their architecture, and other significant details.

Echo: [7] was developed to cope with high performance sharing of data, e.g. visualisations [57]. It is predominantly a content-based system although there are provisions for the subscriber to perform filters. It is based on a peer-to-peer network and matching is performed on the subscriptions.

Gryphon: [8] has been developed for the distribution of large volumes of data in real-time with thousands of clients on a public network [54,58]. It is a content-based system with a client-server architecture where the servers are organised into fully connected cells with redundant links between cells. It uses a matching tree algorithm with expressions over the attributes.

JEDI: has been developed as an internet wide object-oriented event notification system for Java [59]. It is a content-based system with a hierarchical client-server architecture. Matching events to subscriptions is achieved through a simple pattern matching algorithm.

[7] `http://www.cc.gatech.edu/systems/projects/ECho/` (April 2007)

[8] `http://www.research.ibm.com/distributedmessaging/gryphon.html` (April 2007)

Scribe: [9] has been developed as a large scale, decentralised event delivery system [60]. It is a subject-based system in a peer-to-peer network providing best-effort semantics for messages. Matching is performed by numeric keys in a look up table and message delivery relies on the Pastry [61] peer-to-peer network system.

SIENA: [10] has been developed as a generic internet scale event notification system [62]. It is a content-based system in a client-server architecture. It uses a variation on the matching tree algorithm.

2.3 Data Streams and Data Stream Processing

Digital streams can be produced by many sources, e.g. audio, video, sensors, and monitoring scripts. The research challenges for audio and video streams tend to focus on quality of service issues, e.g. ensuring that smooth playback is possible while delivering a video stream to a user in real time [63]. Those for streams from sensors and monitoring scripts focus on processing the data in the stream. The subject of this thesis is making streams of data available for processing and this will be the focus of the subsequent discussion.

Processing data as a stream is a new paradigm for handling data. A data stream is an append only, time varying, unbounded, sequence of data [15]. Streams of data appear in many different situations, e.g. financial applications [1, 2], sensor networks [3, 4], and monitoring information [5–11]. These stream applications will each have their own characteristics, e.g. data may arrive in bursts or at a regular frequency, and the users will have differing demands, e.g. the latest stock price or looking for patterns of attack in network monitoring data. The characteristics of data streams introduce interesting new research questions such as:

- How can a data stream be modelled?
- How can a data stream be queried?
- How can a data stream that is infinite be processed?

[9] `http://research.microsoft.com/~antr/SCRIBE/` (April 2007)
[10] `http://www-serl.cs.colorado.edu/~carzanig/siena/` (April 2007)

- How can the history of an infinite stream be stored in bounded storage?

These have started to be addressed by the research into data stream systems, the results of which will be outlined below.

2.3.1 Data Stream Management Systems

While some of the use cases for storing, managing, and querying data streams are similar to those for a database management system (DBMS), there are some significant differences. In particular, how a data stream system should cope with updates to the data.

A DBMS presents a consistent, unchanging instance of a database for the duration of a transaction, even when the database is being updated rapidly. This means that the operations in a transaction are unaware of the changes made to a database by another transaction that is running in parallel. On the other hand, users of streaming data are often interested in receiving the "freshest" data possible, i.e. they are interested in the changes to the data. Therefore, techniques for storing, managing, and querying data streams are being developed with the nature of streams in mind resulting in data stream management systems (DSMS) [15, 16].

Some centralised DSMSs have already been developed and implemented, e.g. Aurora [64], STREAM [65], TelegraphCQ [66], mostly in research projects in the United States of America. These systems support the querying and management of streams. Figure 2.3 [16] presents an abstract architecture for a DSMS. It consists of components for handling input data streams, processing and querying those streams, and outputting answer streams.

The input monitor is responsible for receiving tuples from the input streams. Typically it will try and receive all data from the streams. However, if the arrival rate is too high for the system to cope with, the input monitor will drop some tuples. The input monitor also maintains statistics about the arrival rates of the streams.

A DSMS will typically have three storage areas. The working storage is responsible for storing those tuples that are required for answering the current queries. The summary storage is used to store summary information about the stream, e.g. the average value for a specific period of time. The static storage is used for both meta

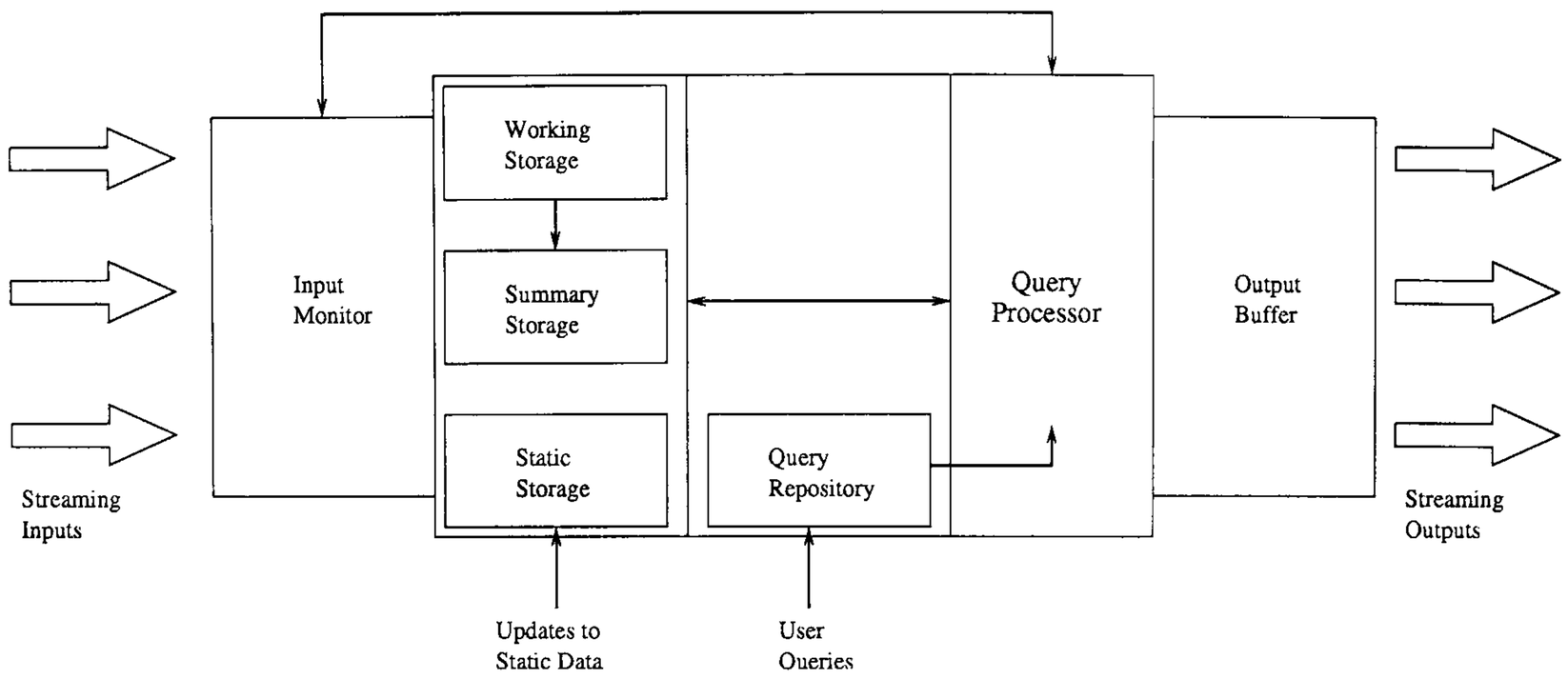


Figure 2.3: Abstract architecture for a Data Stream Management System.

information about the streams and for storing static, or stored, information, e.g. the locations of the sensors that generate the data streams.

The DSMS also stores the queries being posed by the users and has a query processor for generating answers to these queries. The query processor communicates with the input monitor so that the query plans can be adapted to suit the arrival characteristics of the streams. There is also an output buffer for streaming the answers to the users.

2.3.2 Queries over Data Streams

A query posed over a database is passed to the query execution engine of the DBMS where it is optimised for the data currently in the database. The query is then evaluated once over the set of data in the database at the instant the query is posed. This model of querying the data currently in the database is not consistent with the characteristics of a data stream, where the data is continuously arriving and the user is interested in the changes in the data. One approach could be to store the data stream to a database and then the user could periodically pose their query over the history of the stream. However, this still does not match the continuous nature of a data stream.

In [27], the authors introduced the idea of a *continuous query*. A continuous query is registered with the DSMS, and every time a tuple t arrives that is an answer to the query, the tuple t would be returned on the answer stream to the query.

There has been a lot of research carried out to devise a suitable query language for processing data streams. It is not a straightforward task to devise a language

which has both a meaningful and well-defined semantics for both streams and stored relations. For example, consider a join being carried out on two streams. When a join is conducted in a database, all of the tuples in the two relations are considered. However, a data stream is unbounded, and an answer must be returned immediately. This has led to the introduction of window operators which limit the scope of the stream considered, either by the number of tuples or in time, to allow a partial join to be performed. Four types of windowing operator have been proposed in the literature.

Snapshot Windows: which define a specific part of a stream in time, i.e. the window has a constant start and end time [67].

Landmark Windows: which define a window starting at a particular moment in time and extends as the stream arrives, i.e. it has a constant start time but variable end time [68].

Sliding Windows: which define a specific size of window that moves in time as the stream is published, i.e. it has variable start and end time but constant size in time or space [68].

Jumping Windows: which also define a specific size of window but it is not recomputed every time a new tuple is added to the stream, the re-evaluation rate of the window is also controlled and as such it jumps along the stream [69].

One approach, used in the Aurora system [64], is to use a procedural language. Specifically, in Aurora a query is constructed by selecting boxes (which represent operators such as selection or join) and then connecting these boxes together with directed edges.

An alternative approach, being followed in the STREAM project, is to devise a declarative query language specifically for streams, something akin to SQL for a relational database. This has resulted in the Continuous Query Language (CQL) [70]. Within CQL there is support for specifying the size of the window, the ability to process both static and stream data, and the ability to control the execution rate and the returned answer, i.e. how it is streamed back.

More recently there has been consolidation between the various stream projects

resulting in the definition of StreamSQL[11] [71]. Like CQL, StreamSQL is based on SQL and is a declarative language. StreamSQL further refines the concepts, e.g. by providing additional windowing constructs, and combines the advantages of the previous approaches. However, the language has not yet been implemented in a stream processing system.

2.3.3 Existing Data Stream Management Systems

The following gives a brief overview of some of the existing DSMSs.

Aurora [12] is a workflow-oriented system working with sensor data [64, 72]. It has a procedural query language where users connect operators with directed edges. Within this query language there is support for a wide variety of window operators but it is only capable of handling streaming data, i.e. there is no support for stored data.

STREAM [13] is a general purpose DSMS that is able to process both stored and stream data [15,65]. The project has modelled data streams using a relational approach and uses a SQL like query language called the Continuous Query Language (CQL) [70]. At present there is only support for one type of window operation.

TelegraphCQ [14] is a continuous query processing system for processing sensor data [66]. The system has support for a wide variety of window operators along with relational operators. The project focuses on adaptable query processing based on the arrival rate of the input streams. Novel data structures have been developed for processing streams including Eddies [73] which are able to adapt their processing on a per tuple basis and Flux [74] which can repartition stateful operators such as a join during query execution.

---

[11] `http://www.streamsql.org` (June 2007)

[12] `http://www.cs.brown.edu/research/aurora/` (April 2007)

[13] `http://infolab.stanford.edu/stream/` (April 2007)

[14] `http://telegraph.cs.berkeley.edu/` (April 2007)

2.3.4 Distributed Stream Processing

The DSMSs presented above have focused on processing data streams at some centralised point. However, by their nature, data streams are often highly distributed. Research has also been conducted in processing data streams in a distributed manner. This section does not address wireless sensor networks as they have their own distinct problems, e.g. power management, unknown communication networks, high failure rates of nodes [75].

Borealis [15] is a distributed stream processing system [18] based on the Aurora system. A Borealis node consists of an Aurora stream processing engine along with a couple of components for dealing with the distributed setting that were initially developed in the Medusa project [76]. The first component keeps track of all of the streams known to the system. This allows a user to construct a query without knowledge of where the stream originates. The second component allows the system to balance the load of the queries across several nodes. Each node is "selfish" in its load management, which it is claimed results in an equal balance across nodes.

dQUOB [16] is a distributed stream processing engine [19]. It uses precompiled trigger queries, called "quoblets", to filter and process streams close to their origin. There is no built-in support for processing static data, although a quoblet can be used to trigger some external function.

StreamGlobe [17] is a Peer-to-Peer system for publishing and querying data streams of astronomical data on a computational Grid [20]. The system only supports continuous queries. Queries are optimised by combining common query operators and pushing operators as close to the data source as possible. The system has no mechanism for storing the history of a data stream and making it available for querying.

---

[15] `http://www.cs.brown.edu/research/borealis/public/` (April 2007)

[16] `http://www.cs.indiana.edu/~plale/projects/dQUOB/` (April 2007)

[17] `http://www-db.in.tum.de/research/projects/StreamGlobe/` (April 2007)

2.4 Summary

This chapter has introduced the research topics of data integration and stream processing. It showed that data integration allows multiple, autonomous, distributed stored data sources to be accessed through a common global schema. That is, the global schema describes a virtual database over which queries can be posed and answers returned by accessing the actual data sources.

Details of techniques for processing streams of data were also presented. Most of this research has been performed in a centralised setting although the techniques are beginning to be applied in a distributed manner. However, there has been no research on how to integrate multiple, autonomous, distributed streams.

The next chapter presents a motivating problem for integrating distributed data streams.

Chapter 3

Motivation: Grid Information and Monitoring Systems

The work of this thesis has been motivated by the problem of providing up-to-date status information about the resources available on a computational Grid. This problem is addressed by Grid information and monitoring systems which encompass Grid information systems, Grid monitoring systems, and unified Grid information and monitoring systems.

Section 3.1 provides a brief introduction to the application area of Grid computing which aims at sharing computational resources from multiple, autonomous organisations. This requires middleware that allows a computational resource to interact on a Grid. A component of any Grid middleware is the information and monitoring system.

The information and monitoring system allows Grid resources, and other Grid middleware systems, to know the existence and status of the available Grid resources. Section 3.2 considers the requirements for a Grid information and monitoring system, i.e. the types of information required by other middleware systems and how this should be presented, along with functional requirements.

Existing approaches to Grid information and monitoring systems are presented in Section 3.3. These existing systems are analysed against the identified requirements for a Grid information and monitoring system.

3.1 Grid Computing

This section introduces the idea of computational Grids, the kind of software that is required to enable a computational Grid, and details of some Grid projects. Grid computing will provide the environment for integrating streams of data.

3.1.1 Computational Grids

A *computational Grid* is a collection of connected, geographically distributed computing resources belonging to one or more different organisations [12, 77]. Typically the resources are a mix of computers, storage devices, network bandwidth and specialised equipment, e.g. supercomputers or databases. A computational Grid provides instantaneous access to files, remote computers, software and specialist equipment [78]. To a user, a Grid behaves like a single virtual supercomputer.

The idea of a computational Grid uses the power grid as a metaphor for sharing computational resources. In the same way as one plugs an electric device into the power grid and gets instant electrical power, a user should be able to "plug" into a computational Grid and gain instant computational power. Thus, a Grid would appear as a single virtual supercomputer comprising the individual computing resources connected to the Grid at any moment in time. The metaphor can be carried along further. Just as electricity in the power grid can be provided by several companies, the computational resources in a computational Grid may be provided by several different organisations. Moreover, it must be possible to "charge" the user of the Grid for use of these resources. The payment may be in the form of money based on the amount that the resource was utilised, or by making their own resources available to other Grid users.

3.1.2 Grid Projects

The concept of a computational Grid has existed since the mid 1990s and has grown out of the distributed and high performance computing communities. There have now been several projects to construct Grids to perform different tasks. These include:

CrossGrid: [1] A European Union project running from 2001 to 2005 with the aim of developing, implementing and exploiting new Grid components for interactive compute and data intensive applications such as visualisations. The project aimed to use components developed in the DataGrid project as well as develop its own tailored Gridware [79].

DataGrid: [2] A European Union project running from 2001 to 2004 with the aim of developing and testing a technological infrastructure and middleware that will cope with the vast amounts of data produced in scientific experiments such as the Large Hadron Collider [80].

EGEE: [3] A continuation of the DataGrid project running from 2004 and anticpated to end in 2008. The emphasis in the EGEE project is on providing a production quality Grid. The EGEE project should result in scientists having 24/7 access to major computing resources across the globe [81].

Grid2003/Grid3: [4] A collaboration beginning in 2003 involving more than 25 sites in the USA and Korea that collectively provides more than 2000 CPUs. Grid3 was able to process more than 700 concurrent jobs from a number of scientific domains [82].

TeraGrid: [5] An American project aiming to build a Grid for scientific research capable of 20 teraflops of computation, distributed across 5 sites in the USA, with 1 petabyte of storage, and a network bandwidth of 40 Gigabits per second [83]. The project began in 2001 and is expected to continue until 2010.

3.1.3 Components of a Grid

Each Grid requires middleware to enable the resources to behave as a virtual computer. Several sets of middleware have been developed. Currently, the two most

[1] `http://www.crossgrid.org` (February 2007)
[2] `http://eu-datagrid.web.cern.ch` (February 2007)
[3] `http://www.eu-egee.org/` (February 2007)
[4] `http://www.ivdgl.org/grid2003` (February 2007)
[5] `http://www.teragrid.org` (February 2007)

widely used are the Globus Toolkit[6] [84], and gLite [7] [85], both of which achieve similar functionality and consist of several services.

The services of the middleware mimic the behaviour of an operating system on a computer. The components of the DataGrid/gLite middleware, and their interactions, can be seen in Figure 3.1 and are similar to those presented in [77].

User Interface: allows a human user to submit jobs, e.g. "analyse the data from a physics experiment, and store the result".

Resource Broker: controls the submission of jobs, finds suitable available resources and allocates them to the job.

Logging and Bookkeeping: tracks the progress of jobs and when jobs are completed informs users as to which resources were used, and how much they will be charged for the job.

Storage Element (SE): provides physical storage for data files.

Replica Catalogue: tracks where data is stored and replicates data files as required.

Computing Element (CE): performs the processing of jobs, taking data from storage elements.

Monitoring System: monitors the state of the components of the Grid and makes this data available to other components.

3.2 Requirements for a Grid Information and Monitoring System

The purpose of a Grid information and monitoring system is to make available to users and other Grid components details about the resources on a Grid, along with their status information. This is separated from the task of capturing monitoring information, which is performed locally at the Grid resource, e.g. computing element,

[6] `http://www.globus.org` (February 2007)

[7] `http://glite.web.cern.ch` (February 2007)

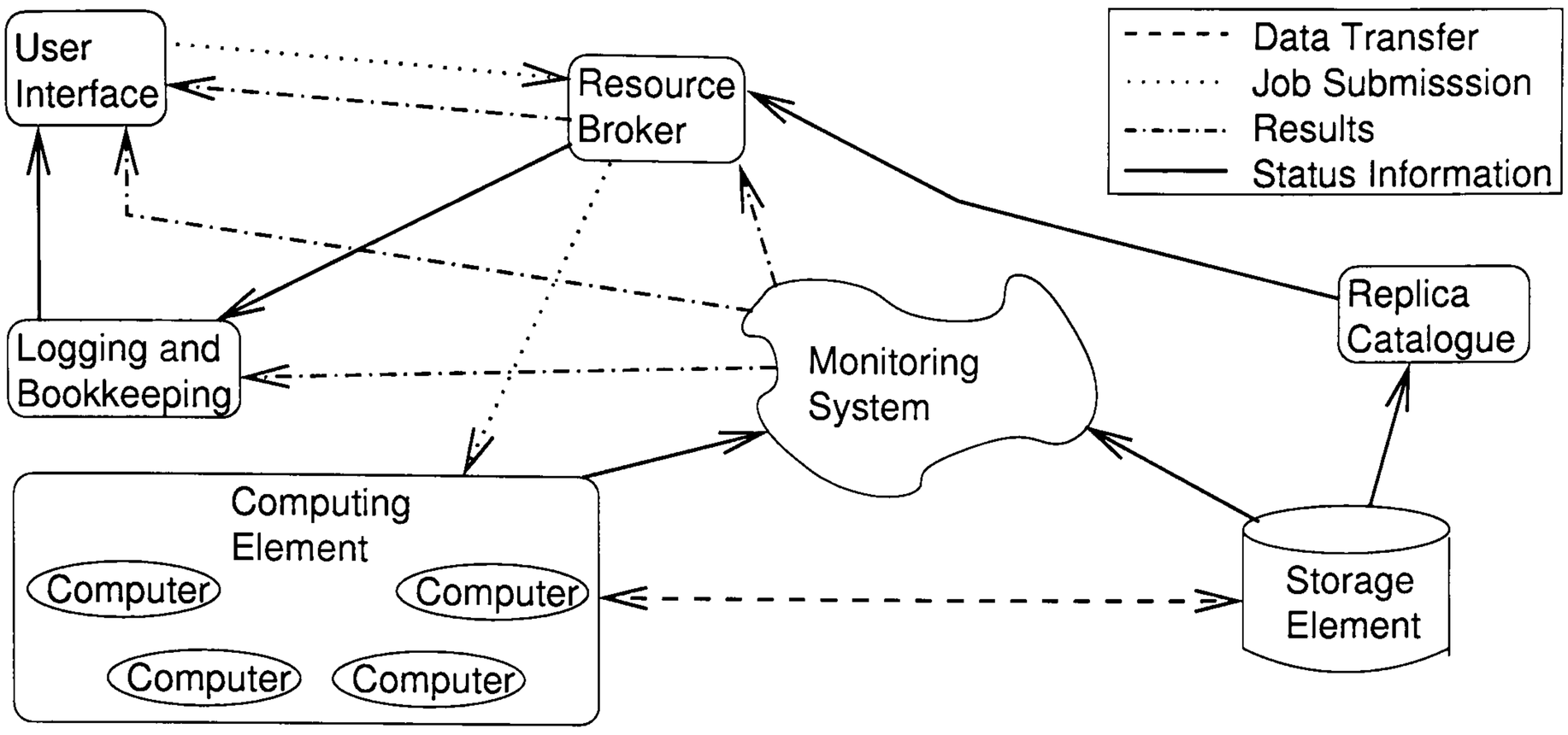


Figure 3.1: The major components of DataGrid/gLite middleware.

storage element or between network nodes, and for which several tools exist [86, 87]. For example, network monitoring measurements can be made using the PingER tool [88] which can be used to measure the throughput between two network nodes. The results of these local monitoring tasks are then made available across the Grid by the information and monitoring system.

As a basis for discussing the requirements that such a system should meet, the following use cases are considered. These use cases were originally published in [89] and similar cases have been considered in [90].

1. A resource broker needs to quickly (within 10 seconds) locate a computing element (CE) that has 5 CPUs available, each with at least 200 MB of memory. The CE should have the right software installed, and the user must be authorised to use it. The throughput to a storage element (SE) needs to be greater than 500 Mbps.

2. A visualisation tool, that allows users to monitor the progress of their jobs, needs to be updated whenever the status of a job changes.

3. Network administrators need to interrogate the past state of the network so that typical behaviour can be ascertained and anomalies identified.

3.2.1 Publishing Data

There are many different kinds of information about a Grid, which come from numerous sources. The following are examples:

- Measurements of network throughput, e.g. made by sending a `ping` message across the network and publishing the measurements (use cases 1 and 3 above);
- Job progress statistics, either generated by annotated programs or by a resource broker (use case 2);
- Details about the topologies of the different networks connected (use cases 1 and 3);
- Details about the applications, licences, etc., available at each resource (use case 1).

This monitoring data can be classified into two types based on the frequency with which it changes and depending on the way in which it is queried:

Stored data (pools): This is data that does not change regularly or data that does not change for the duration of a query, e.g. data that is being held in a database management system with concurrency control. Typical examples are data about the operating system on a CE, or the total space on a SE (use case 1).

Dynamic data (streams): This is data that can be thought of as continually changing, e.g. the memory usage of a CE (use case 1), or data that leads to new query results as soon as it is available, for example the status of a job (use case 2).

A core requirement of a Grid information and monitoring system is that it should allow both stored and streaming data to be published. The act of publishing involves two tasks:

1. Advertising the data that is available, and
2. Answering requests for that data.

3.2.2 Locating and Querying Data

Grids typically consist of a number of geographically distributed sites where each site has several resources, e.g. clusters of computers and storage devices, which are connected by high speed LAN connections. Each resource would be instrumented with scripts to publish their status information. The sites on a Grid can be located around the world and are connected by wide-area network links. For example, the Large Hadron Collider computational Grid consists of resources provided by over 175 research institutions located in Asia, Europe, and North America[8].

As such, data about Grid resources will be scattered across the entire fabric of the Grid. The information and monitoring system must provide mechanisms for users of the Grid to locate data of interest. In addition, users need a *global view* over the data published in order to understand relationships between the data and to query it.

Since a Grid information and monitoring system should be able to publish both stored and streaming data it should also be possible to query both types of data, either separately or in a combined manner. It should be possible to ask about the state of a stream right now (a *latest-state* query—use case 1), continuously from now on (a *continuous* query—use case 2), or in the past (a *history* query—use case 3).

Up-to-date answers should be returned quickly, e.g. in use case 1 the resource broker requires that the data is no more than a few seconds old. To be accepted by users, the query language should capture most of the common use cases, but should not force a user to learn too many new concepts.

3.2.3 Scalability, Robustness, and Performance

A Grid is potentially very large: in February 2007, the Large Hadron Collider Grid[9] had 32,412 CPUs available, located at 177 sites throughout the world, each producing status information. In the normal use of a Grid, the fabric will be unreliable: network connections will fail and resources will become inaccessible.

It is important that the information and monitoring system can *scale*. It needs to be able to handle a large number of sources, publishing potentially large amounts of

[8] `http://goc.grid-support.ac.uk/gridsite/monitoring/` (February 2007).

[9] `http://goc.grid-support.ac.uk/gridsite/monitoring/` (February 2007).

data. Likewise there will be a large number of users of monitoring information, both humans and Grid components, who require correct answers in a timely manner. The information and monitoring system should not become a performance bottleneck for the entire Grid. It should be able to cope with large numbers of queries received at the same time.

The information and monitoring system itself should be resilient to failure of any of its components, otherwise the whole Grid could fail along with it. The information and monitoring system cannot have any sort of central control as resources will be contributed by organisations that are independent of each other.

3.2.4 Security

Users of the Grid may only use resources for which they are authorised. Only if they authenticate themselves should they be granted access to those resources. The Grid information and monitoring system should respect these authorisation rules and only provide a user information on resources they are allowed to use.

Similar mechanisms need to be in place to authenticate the sources that wish to publish data. For example, if a rogue data publisher were able to say that it was always lightly loaded it could distort the distribution of jobs by a resource broker. The data sources must also respect the authorisation rules that have been imposed and only pass data on to those who are authorised to "see" it.

3.3 Existing Systems and Possible Approaches

This section considers the possible approaches that could be followed in developing a Grid information and monitoring system. Several existing Grid monitoring, Grid information, and unified Grid information and monitoring systems are also considered to see if they meet the requirements identified in Section 3.2.

3.3.1 Possible Approaches

Chapter 2 provided details of publish/subscribe systems, data stream systems, and data integration systems. However, none of these existing types of systems nor stan-

dard database systems are suitable for the requirements of a Grid information and monitoring system.

It has been suggested in [91] that an information and monitoring system could be implemented as a relational database. The database would store all data about the status of the Grid and the users would be able to query it. Database systems already provide the security mechanisms required for a Grid. However, a DBMS does not meet many of the other requirements identified. For example, it takes time to load all the data into a single repository and this would require a large amount of storage space. On top of this, the database would be a single point of failure. If the database became uncontactable then no monitoring information would be available. Additionally, this approach would not scale to the size required for a typical Grid and would be unable to cope with the streaming nature of the data.

While publish/subscribe systems (Section 2.2) provide a mechanism to pass messages from publishers to subscribers without either needing to know the details of the other, there is no inbuilt mechanism to store and query historical data. Additionally, the fact that monitoring data will be streaming does not meet the periodic publishing for which publish/subscribe systems are designed.

Most of the data stream systems (Section 2.3.3) developed to date are centralised and so have many of the same problems as the database approach, i.e. a central point of failure. The distributed data stream systems (Section 2.3.4) while overcoming the central point of failure problem, still do not meet all of the requirements. For example, the dQUOB system [19] must pre-compile all of the continuous queries and so is less able to deal with the dynamic nature of a Grid with ad hoc queries. Additionally, these systems do not support queries over the past state of the system.

While following a data integration approach (Section 2.1) overcomes the problems of using a single database, and can be made to provide security by imposing a suitable mediation mechanism, there is currently no system or theory that can integrate streams of data.

So far, the existing systems and approaches considered in Chapter 2 do not meet the requirements identified in Section 3.2 for a Grid information and monitoring system. However, an approach combining the ideas of data streams, publish/subscribe systems, and data integration could provide suitable functionality.

3.3.2 Proposed Theoretical Architectures

In the literature there have been two proposals for a generic model for a Grid information and monitoring system. The first was the Grid Monitoring Architecture (GMA) which was proposed as a general approach to Grid monitoring. The second was the Generic Monitoring and Information System Model which has been proposed for the purpose of comparing the performance of three existing systems.

Grid Monitoring Architecture

The Grid Monitoring Architecture (GMA) was proposed by Tierney *et al.* [92] and has been recommended by the Global Grid Forum, now the Open Grid Forum[10], for its scalability. It is a simple architecture comprising three main types of actors:

Producers: Sources of data on the Grid, e.g. a mechanism to allow a sensor to publish its readings, or a description of a network topology.

Consumers: Users of data available on the Grid, e.g. a resource broker, or a system administrator wanting to find out about the utilisation of a Grid resource.

Directory Service: A special purpose component that stores details of producers and consumers to allow consumers to locate relevant producers of data.

The interaction of these actors is schematically depicted in Figure 3.2. A producer informs the directory service of the kind of data it has to offer. A consumer contacts the directory service to discover which producers have data relevant to its query. A communication link is then set up directly with each producer to acquire data. Consumers may also register with the directory service. This allows new producers to notify any consumers that have relevant queries.

The GMA also proposed an *Intermediary* component that consists of both a consumer and a producer. An intermediary may be used to forward, broadcast, filter, aggregate or archive data from the producers. The intermediary then makes this data available to the consumers from a single point in the Grid.

By separating the tasks of information discovery, enquiry, and publication, the GMA is *scalable*. However, it does not define a data model, query language, or a

[10] `http://www.ogf.org` (February 2007)

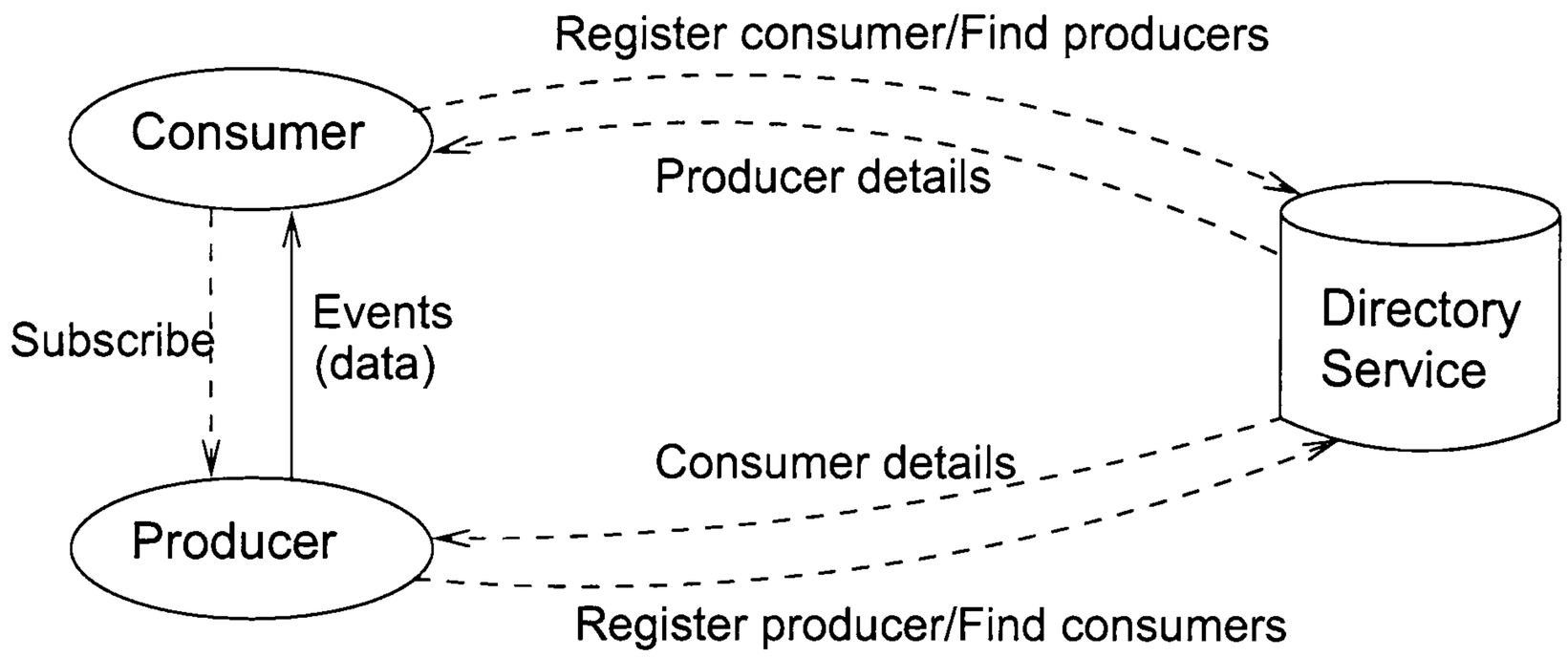


Figure 3.2: The components of the GMA and their interactions. (The dashed lines depict the exchange of control messages whilst the solid line shows the flow of data.)

protocol for data transmission. Nor does it say what information should be stored in the directory service. There are no details of how the directory service should perform the task of matching producers with consumers.

Generic Monitoring and Information System Model

The Generic Monitoring and Information System Model was proposed by Zhang *et al.* [90,93] to allow them to compare the performance of three monitoring and information systems. It is a simple architecture consisting of four components:

Information Collector: Sources of data about a single aspect of a resource on the Grid, e.g. a sensor, probe, or simple program to generate data describing some property of a Grid resource. A single Grid resource will run several information collectors to provide data about different aspects of the resource.

Information Server: Collects data from several information collectors. The information server presents a snapshot of all of the data available about a single Grid resource.

Aggregate Information Server: Collects and aggregates information from several information servers.

Directory Server: Provides details of all the components of the information and monitoring system along with data published about the status of the resources on a Grid.

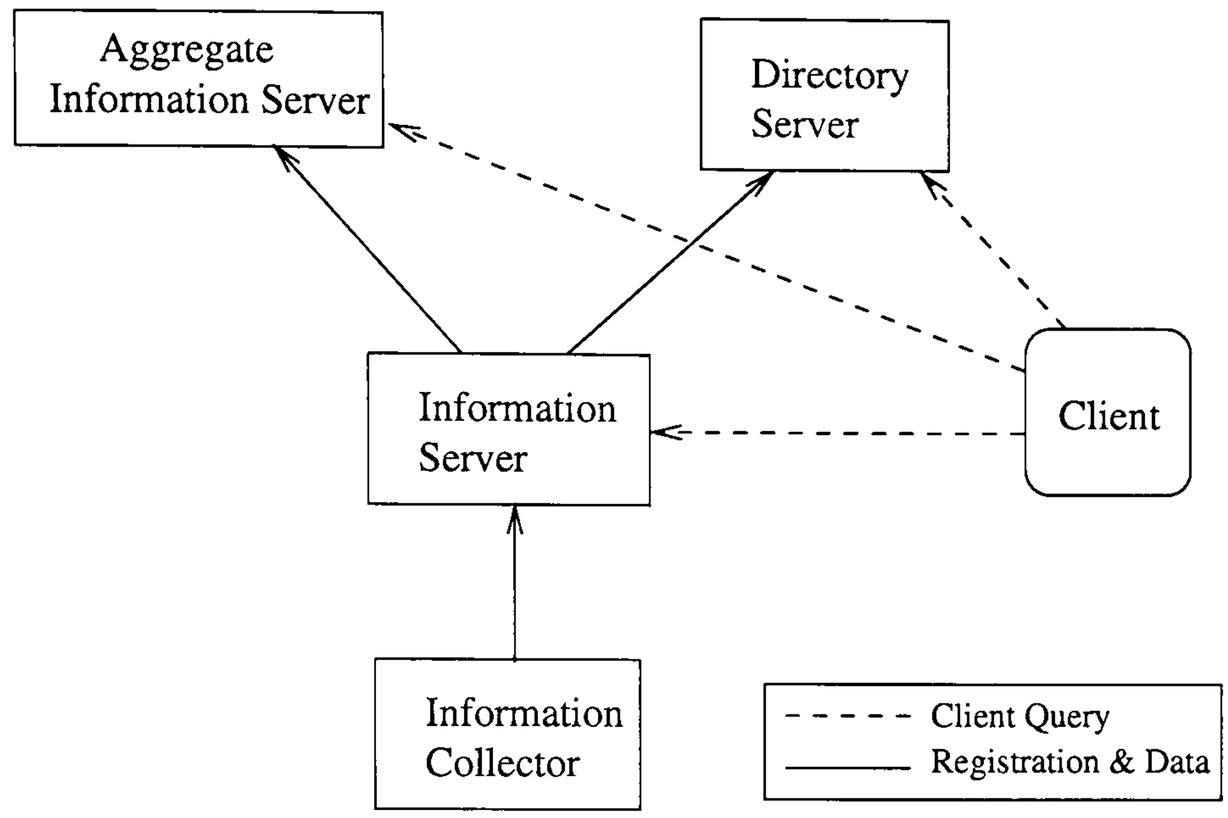


Figure 3.3: The components of the Generic Monitoring and Information System Model and their interactions.

The interaction of the components of the Generic Model are depicted in Figure 3.3. A client may query the information server, aggregate information server, or the directory server to retrieve data about the status of the Grid.

The idea of the information server is that it collects all the monitoring data about a single Grid resource. The aggregate information servers would then collect all of the data of the resources of a single site and make this data available.

The model fails to separate out the tasks of discovering where data is published and retrieving that data. Both of these tasks are provided by the directory server which stores details of the information providers in the monitoring system together with the actual monitoring data. As a Grid scales to larger numbers of resources, this could become a problem as the directory server may not be able to handle the volume of data and become a performance bottleneck. The lack of separation also begs the question, what is the purpose of the aggregate information server if this functionality is built into the directory server?

Like the GMA, the Generic Monitoring and Information System Model does not provide any implementation details such as a data model, communication protocol, or how to perform the matchmaking of client queries to monitoring data.

3.3.3 Existing Systems

A number of Grid monitoring, Grid information, and unified Grid information and monitoring systems have been developed. Each has been developed with a different emphasis and aim. Some have been tailored to meet the requirements of a specific Grid, e.g. CODE [94] for the NASA Information Power Grid [95], while others are prototype implementations to prove the correctness of an approach, e.g. PYGMA [96] for the GMA.

Many Grid monitoring systems only provide specific types of information. For example, Autopilot [97] used in the GRADS project [98] and G-PM/OCM-G [99] in the CrossGrid project [79] have been developed to track the progress of jobs running on the Grid, while systems such as the Network Weather Service (NWS) [9, 100] have been developed to monitor the status of the network resources on the Grid.

Systems such as SCALEA-G [101], Mercury [102], and the Monitoring and Discover Service/System[11] (MDS) [103, 104] of the Globus Toolkit [84] have been designed as unified Grid information and monitoring systems so that they can cover all aspects of data about the Grid. These can be seen to be implementations of the GMA, with SCALEA-G and MDS both using standard data models.

SCALEA-G uses XML as a data format and makes use of the XPath and XQuery query languages. It consists of sensor managers, a client service, and a directory service. It was originally developed as a parallel machine monitor. So far it has only been deployed on a small testbed with no indication of how it will scale to a large Grid.

MDS is the most widely used of the existing monitoring systems, and has gone through several releases. MDS 2 [103] has been widely deployed throughout the world and uses the Lightweight Directory Access Protocol (LDAP) [105] as its data model and query mechanism. MDS 3 moved to an XML data format but was only deployed in a limited number of situations. The latest release, MDS 4 [104] has continued using XML as its data format and has been provided as a set of Web Services. Further details of MDS are provided below.

A comprehensive comparison of systems that can provide monitoring data or in-

[11] Versions 1, 2, and 3 were called the Monitoring and Discovery Service. As of version 4, it is now called the Monitoring and Discovery System.

formation about a Grid can be found in [106].

Monitoring and Discovery Service/System (MDS)

The main components of MDS are:

Information Providers: Publish monitoring data about one aspect of a Grid resource.

Information Services: Collect all the data about one Grid resource.

Aggregate Directories: Hierarchically organised to collect all the data about resources at one site, one virtual organisation, etc.

Client: An interface, e.g. browser or program, through which a user can pose queries.

The aggregate directories at the site level in the hierarchy forward their data to other aggregate directories at higher levels e.g. the virtual organisation level. It can be seen that MDS implements the GMA. An information provider plays the role of a producer, a client plays the role of consumer, an aggregate directory plays the role of directory server, and both aggregate directories and information servers play the role of intermediaries.

Data is also organised hierarchically in a structure that provides a name space, a data model, wire protocols and querying capabilities. MDS 1 and 2 were based on the LDAP data model and query language. In MDS 3, the data model was changed to XML with support for the query languages XPath and XQuery. The latest version continues to use an XML data model but queries are now posed using Web Services Notification (WS-N) [107] although there is still support for XPath and XQuery.

Although the hierarchical architecture makes it *scalable*, MDS does not meet other requirements outlined in Section 3.2. Firstly, hierarchical query languages have limitations. For one, the hierarchy must be designed with popular queries in mind. Moreover, there is no support for users who want to relate data from different sections of the hierarchy—they must process these queries themselves.

Secondly, to be able to offer a global view of the Grid to users, a hierarchy of aggregate directories must be set up manually—information providers, information servers and aggregate directories need to know which information server/aggregate

directory further up the hierarchy to register with. The system does not automate this, nor does it recover if any component in the hierarchy fails.

Lastly, MDS 2 only supports queries for the most recent information with no assurance that the answers are up-to-date. Support for continuous queries has been provided in MDS 3 and 4 but it is unclear how well it is able to handle streams of data.

It has also been left up to the user to create and maintain archives of historical information by (i) storing the various latest-state values that have been published via MDS in a database and by (ii) providing an interface to allow the system to access the database. This approach would require considerable effort on the side of the user.

3.4 Summary

This chapter has introduced the application area of Grid computing which aims to share computational resources from multiple organisations as if they were part of a virtual supercomputer. This requires Grid middleware which allows a computational resource to interact with other resources on the Grid.

At the heart of any Grid middleware is the information and monitoring system. The requirements for this system were identified. Existing approaches and systems were considered against these requirements and shown not to meet all of them in their separate ways. It was argued that an approach involving the integration of data streams would be appropriate.

The next chapter will introduce a generic architecture for integrating streams of data that will meet the requirements of a Grid information and monitoring system.

Chapter 4

A Stream Integration System

This chapter proposes an architecture for a stream integration system that can incorporate both streaming and stored data sources. While the architecture has been motivated by the problem of a Grid information and monitoring system, the design itself is generic and could be used in any scenario where there is a need to integrate distributed sources.

Section 4.1 provides the details of the architecture for a stream integration system. In Section 4.2 details of the Relational Grid Monitoring Architecture (R-GMA), which is a partial implementation of the proposed architecture, will be given. Finally, Section 4.3.2 will compare the stream integration approach with some of the existing systems detailed in Section 3.3.3.

4.1 An Architecture for a Stream Integration System

This section presents the architectural design for a stream integration system. The system is based on the relational data model. It applies the ideas for integrating sources of stored data to sources of stream data and allows the two types of data to be mixed. The focus of the design is on the ideas and rationale which will draw upon the motivating application of a Grid information and monitoring system. Details of the theory required to realise the architecture will be presented in Chapters 5 and 6. Details of the architecture have previously been published in [89, 108, 109].

4.1.1 A Virtual Dataspace

As stated in the requirements for a Grid information and monitoring system (Section 3.2), users need to be able to locate data of interest using the information and monitoring system. The difficulty is that data is scattered across the whole Grid. It would be useful if the system could operate like a database, presenting a schema over which queries can be posed. However, as already stated, it would not be practical to stream all data into a central database, as a database management system (DBMS) would introduce delays in answering queries while it waits for data to load and the DBMS would become a single point of failure for the Grid. Therefore, the stream integration system should create the illusion of a dataspace through which all the data is available. This *virtual dataspace* would allow access to both the streaming and stored data in the system.

The techniques of data integration, detailed in Section 2.1, allow a virtual database to be created from a set of distributed source databases. Such data integration systems use a mediator [23] for matching queries posed over the global schema with sources of relevant information. The proposed stream integration system follows a *local-as-view* approach to data integration, where data sources describe their content as views on the global schema. This provides the flexibility of being able to add and remove sources without reconfiguring the global schema although query answering is not straightforward.

The techniques in the literature on data integration can only handle stored data sources. The stream integration system needs to be able to publish and integrate both streams of data and stored data, to present a virtual dataspace. A theoretical model and techniques for integrating data streams have been developed and will be discussed in Chapters 5 and 6.

4.1.2 Roles and Agents

The stream integration system takes up the generic consumer and producer metaphors of the Grid monitoring architecture (GMA) [92] and refines them. The stream integration system would allow a client to *play the role* of an information *producer* or a *consumer*. The components of the stream integration system and their interactions

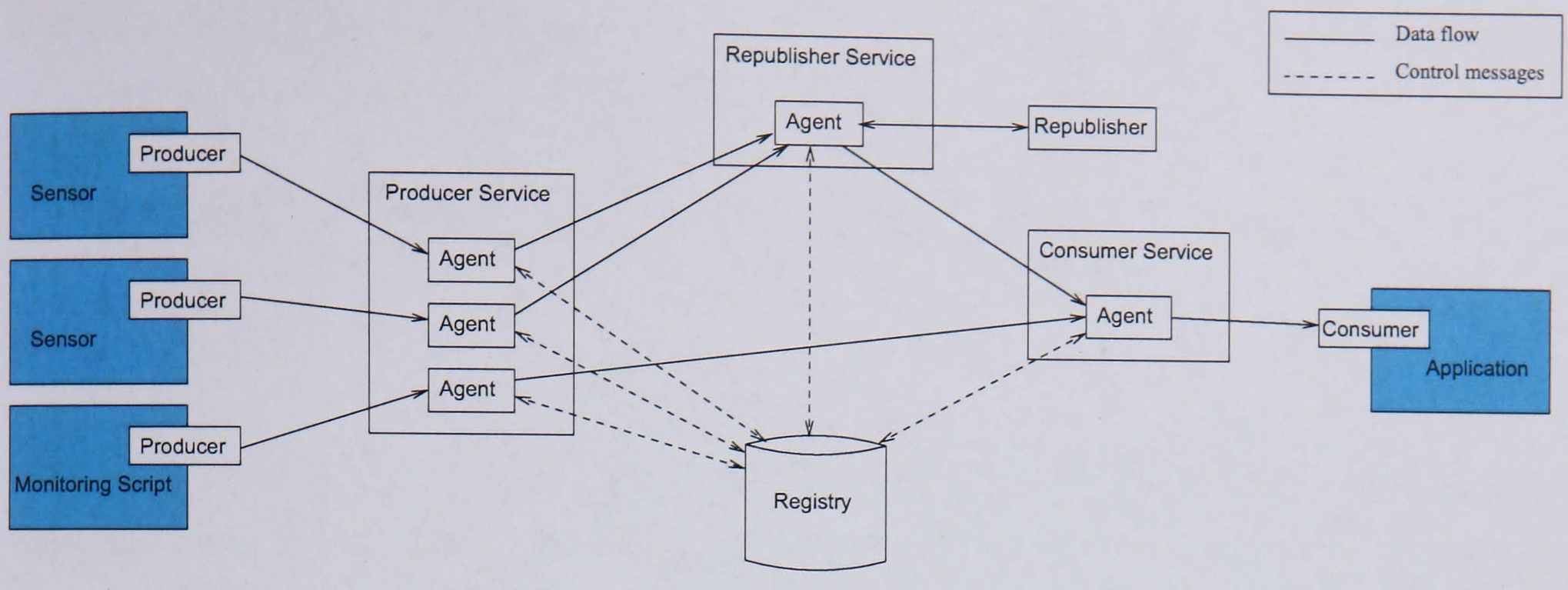


Figure 4.1: The roles and agents of the stream integration system design along with their interactions.

are shown in Figure 4.1. A discussion of these is provided below.

Producers

In order that both stored data and stream data can be published, two producer roles should be supported: a *database producer* and a *stream producer*. A database producer publishes a collection of relations maintained in a relational database, each of which complies with the schema of a specific *stored relation*. A stream producer publishes a collection of streams, each of which complies with the schema of a specific stream relation. The stored or streamed relations of a producer are referred to as its *local relations*.

A producer advertises its local relations by describing them as views on the global schema (Section 4.1.3) which is split into stored and stream relations.

Consumers

A consumer is defined by a relational query. If the query is posed over stream relations, then the consumer has to declare whether it is to be interpreted as a:

Continuous query: a long-lived query that returns each tuple t that satisfies the query condition as it is inserted,

History query: a one-off query that returns those tuples that have previously occurred on a data stream which satisfy the query condition, or

Latest-state query: a one-off query that for the most recent tuple of each of the key values returns the tuple if it satisfies the query condition.

A query over the stored relations is interpreted as a normal database query.

Agents

The stream integration system architecture provides *agents* that allow clients to play the role of an information producer or consumer without needing specific knowledge about the system. All of the knowledge required to play one of the roles is provided by the agent. For example, an application wishing to play the role of a consumer would communicate through a defined interface[1] with an agent. The agent would act on behalf of the application to retrieve the required data and pass it back to the application. It is likely that an implementation would make a set of services available to expose the functionality of the agents.

4.1.3 Global Schema

To interact with each other, producers and consumers need a common *language* and *vocabulary*, in which producers can describe the information they supply and consumers the information for which they have a demand. For the designed system, both the language for announcing supply and the one for specifying demand—the query language—will be based on SQL, extended as needed to cope with the streams of data. The vocabulary consists of relations and their attributes that make up a global schema, which is stored in the *schema service*. The schema service has been omitted from Figure 4.1 for clarity of presentation, as all of the agents and the registry service need to interact with the schema.

Ideally the global schema distinguishes between two kinds of relations, *stored* and *stream* relations. The two sets are disjoint. It should be possible to add and remove relations from the schema as appropriate. Also, if the system is being installed for a particular application domain then it should contain a suitable set of core relations that exist during the entire lifetime of the installation. For example, in the Grid information and monitoring example the schema service would consist of the relations

[1]Either an API or a Web service interface.

required to model the Glue Information Model (aka Glue Schema) for Grid resources [110].

The attributes of a relation will have types as in SQL. In addition to its declared attributes, every stream relation has an additional attribute timestamp, which is of a type DateTime and records the time a tuple was published.

For both kinds of relations, a subset of the attributes can be singled out as the *primary key*. Primary keys are interpreted as usual: if two tuples agree on the key attributes, and the timestamp in the case of a stream relation, then they must also agree on the remaining attributes. However, since data will be published by independent distributed producers, the constraint cannot be enforced.

For stream relations, the keys play an additional semantic role. The key attributes specify the parameters of a reading, i.e. they identify "where" and "how" a reading was taken. The rest of the attributes, except the timestamp, are the *measurement attributes*, i.e. the attributes that state "what" the current reading is. The timestamp attribute identifies "when" a reading was taken.

For instance, in the Grid information and monitoring example a relational version of the Glue information model would contain the stream relation ntp for publishing readings of the throughput of network links. The relation has the schema

$$\mathsf{ntp}(\underline{\mathsf{from}}, \underline{\mathsf{to}}, \underline{\mathsf{tool}}, \underline{\mathsf{psize}}, \mathsf{latency}, [\mathsf{timestamp}]), \tag{4.1}$$

which records the time it took (according to some particular tool) to transport packets of a specific size from one node to another. All attributes except latency make up the primary key of ntp. The types of the attributes in the example have been omitted as they are not important for the understanding of the system.

Intuitively, a specific set of values for the key attributes of a stream relation identify a *channel* along which measurements are communicated. For example, for the ntp relation with the tuple

$$(\texttt{'hw'}, \texttt{'ral'}, \texttt{'ping'}, 256, 93, \texttt{2006-03-17 14:12:35}). \tag{4.2}$$

measuring a latency of 93 ms for a 256 byte ping message between Heriot-Watt University and Rutherford Appleton Laboratory on Wednesday 17 March 2006 at 2:12 pm. the channel is identified by the values

$$(\texttt{'hw'}, \texttt{'ral'}, \texttt{'ping'}, 256). \tag{4.3}$$

Consumers pose queries over the global schema. For example, consider the file `myData` which contains experimental data and is replicated on several storage elements on a Grid. A user would be interested to know how long it will take, based on the current network traffic, to copy the file from the various storage elements where it is replicated to a `workerNode` where the file will be processed, so that a suitable copy can be chosen. Suppose also the global schema contains the ntp relation defined in (4.1) and a stored file allocation relation

$$\mathsf{fat}(\underline{\mathsf{site}}, \underline{\mathsf{file}}, \mathsf{size}, \mathsf{date}), \tag{4.4}$$

which tracks which files are stored at which sites along with their size and date of last modification.

Using these relations we can gather the required information with the SQL-like query

```
SELECT LATEST N.from, N.psize, N.latency
FROM   ntp as N, fat as F
WHERE  N.from = F.site AND
       F.file = 'myData' AND
       N.to = 'workerNode' AND
       N.tool = 'ping'
```
(4.5)

which asks for the sites where the file is stored and the most up-to-date information about the network throughput, based on the ping tool, from those sites to the cluster that will perform the processing. The query uses the keyword "LATEST", which indicates that this is a latest-state query (see Section 4.1.4 for more details on temporal query types). This information can then be used to calculate which will be the fastest site to transfer the file from.

Similarly, producers are able to describe their local relations as views on the global schema.

4.1.4 Producers and Consumers: Semantics

The following will discuss the semantics when producers declare their content using views *without projections*. Each producer contributes a set of tuples to each global relation. Although, selection queries are simpler than the situation considered in data

integration for databases, see Section 2.1, already many important issues arise in the stream setting for this case requiring a theoretical framework to be developed so that query plans for generating answer streams can be computed, see Chapter 5. This case also matches the current requirements of users of the Grid information and monitoring system that has motivated this work.

An intuitive semantics for an instance of the stream integration system would be: a stored relation is interpreted as the union of the contributions published by the database producers; a stream relation is interpreted as a global stream obtained by merging the streams of all the stream producers.

A *stored query* is interpreted over the collection of all stored relations, while a *continuous query* is conceptually posed over the virtual global stream. A *history query* refers to all tuples that have ever been published in the stream or some period defined in the query. Finally, a *latest-state query* posed at time τ_0 refers to the set of tuples obtained by choosing from each active channel the most recent tuple published before or at time τ_0.

Actually, the semantics of stream relations is not as well-defined as it may seem because it does not specify an order for the tuples in the global stream. Since the sources of stream data will be distributed, there can be no guarantee of a specific order on the entire global stream. However, the stream integration system does require that the global streams are *weakly ordered*, i.e. for a given channel the order of tuples in the global stream is consistent with the timestamps. This property ensures that aggregation queries on streams that group tuples according to channels have a well-defined semantics. Chapter 5 will explain how one can enforce weak order on an instance of the global stream.

The suggested semantics of stream relations causes difficulties for some kinds of queries, for instance, aggregate queries over sliding windows where the set of grouping attributes is a strict subset of the keys. In such a case, different orderings of a stream can give rise to different query answers. This issue has not been considered yet.

Among the three temporal interpretations of stream queries, only continuous queries will be supported by default by a stream producer agent. However, when creating a stream producer, it should be possible to instruct the agent to maintain a pool with the history and/or the latest-state of the stream. This would enable it to

answer queries of the respective type. The creation of these pools is optional because their maintenance will impact on the performance of the stream producer agent.

4.1.5 Republishers

A key component of the proposed architecture for a stream integration system is the *republisher*. The idea behind the republisher is that it resembles a materialised view in a database system, which also corresponds to the intermediary in the Grid Monitoring Architecture (GMA) [92]. Their main usage is to reduce the cost of certain query types, e.g. continuous queries over streams, or to set up an infrastructure that enables queries of that type in the first place, e.g. latest-state or history queries.

A republisher combines the characteristics of a consumer and a producer. It is defined by one or more queries over the global schema and publishes the answers to those queries. The republishers can be used to compute partial answers that can then be used to answer other queries. However, the republishers introduce redundancy in the information available, meaning that there can be several possibilities to answer a query. Chapter 5 describes how this is taken into account in the construction of query execution plans for the type of consumer queries considered in this thesis.

Similar to a stream producer agent, a republisher agent which is posing a continuous or mixed query can be configured to additionally maintain a pool of latest-state values or a history. This allows the republisher to also answer the corresponding query type.

Since both input and output to a republisher that is posing a continuous or mixed query are streams, *hierarchies* of republishers over several levels can be built. An important usage for such hierarchies is to bundle small flows of data into larger ones and thus reduce communication cost.

Stream producers often publish data obtained from sensors, such as the throughput of a network link measured with a specific tool. While such primary flows of data, to elaborate on the metaphor, tend to be trickles, with republishers these can be combined into streams proper. For instance, in the Grid scenario considered, there may be a republisher at each of the sites that is used to collect data about the network traffic from that site. Then, at the next level up, there could be republishers collecting all the information between the sites belonging to an entire organisation participating

in a Grid.

Thus a consumer asking for network throughput on all links from a particular site need only contact the republisher for that site, or for the organisation, instead of all the individual stream producers. The mechanisms developed to support such hierarchies are presented in Chapter 6.

The theory developed so far requires that the queries have to be continuous and that these can be used to set up archives of historic or latest-state data. However, in general republishers should be able to support combinations of continuous and stored query types in order to materialise partial answers that can be exploited when answering consumer queries.

4.1.6 The Registry

Producers and republishers are collectively referred to as *publishers*. Consumer agents need to find publishers that can contribute to answering their query. This is facilitated by the *registry*, which records all publishers and consumers that exist at any given point in time.

When a new publisher is created, its agent contacts the registry to inform it about the type of that publisher and, if appropriate for its query type, whether it maintains latest-state or history pools. If the publisher is a producer, the agent registers its local relations together with the views on the global schema that describe their content. If it is a republisher, the agent registers its queries. Similarly, when a consumer is created, the consumer's agent contacts the registry with the consumer's query.

The registry co-operates with the consumer agent in constructing a query plan. It identifies publishers that can contribute to the answers of that query, called the *relevant* publishers.

When considering a continuous or mixed query, and there are republishers present, it is not straightforward to identify which combination of publishers should be used to answer the query. This is because republishers introduce some redundancy among the relevant publishers as they publish the same data as the producers.

In a Grid information and monitoring system, it is important that duplicate data is not returned in answer to a query. Additionally, since republishers themselves pose a continuous query, there is the possibility that the hierarchy of republishers contains

a loop. Full details of the problems of republishers being used in a continuous query and how these may be overcome are discussed in Chapter 6.

When a consumer registers a continuous query, the registry and the consumer agent ensure that during the entire lifetime of the consumer it can receive all the data the query asks for. This requires that when a new producer registers, the registry should have some mechanism to identify those consumers to which this producer is relevant and should notify their agents. The agents will then need a mechanism to integrate the new producer into their query plan.

Details of an approach involving republishers are discussed in Chapter 6. Consumer agents also need to be informed when a republisher goes offline because then the consumer may miss data that it has received via that republisher. Similarly, the registry has to contact a consumer agent if a new relevant republisher is created and when a producer goes offline.

4.2 R-GMA: A Partial Implementation

The Relational Grid Monitoring Architecture[2] (R-GMA) [89, 108, 109, 111] partially implements the stream integration system detailed above. R-GMA was initially developed as part of the EU DataGrid project [80] and continues to be developed in the EU EGEE project [81]. The aim behind R-GMA is to provide a unified information and monitoring system for the Grid which meets the requirements identified in Section 3.2. R-GMA is deployed in several Grid projects including the Large Hadron Collider Computational Grid.

4.2.1 The R-GMA Architecture

R-GMA follows the local as view approach to stream integration and presents a virtual global stream through which the data flows. It does not explicitly support stored relations. An installation of R-GMA comes with the relations of the Glue schema [110]. However, it is possible to add and remove other relations as required.

Together with a Registry Service and a Schema Service, the R-GMA system consists of instances of four main components. These components are supported in their roles

[2] `http://www.r-gma.org/` (February 2007)

by services which correspond to the agents in the stream integration system. The types of components are:

Primary Producer: Allows a stream of monitoring information to be published according to a selection query over the global schema. The primary producer service can be configured to store the history or latest-state information about the stream that it publishes. The primary producer corresponds to the stream producer in the stream integration system presented above but with no support for publishing stored relations.

Secondary Producer: Poses one or more continuous queries over the global schema and is used to maintain the histories or the latest-state of the streams. The secondary producers cannot be used to answer continuous queries.

On-Demand Producer: Allows data to be published that cannot be streamed into the system. In R-GMA, the on-demand producer can be used to answer *static queries* that resemble SQL queries to a database. The on-demand producer application must provide query answering capabilities to retrieve the required information which the supporting service can then provide to the consumer as an answer. There is no support to link the data in a specific on-demand producer with the data in any other producer. There is no corresponding component in the stream integration system although there are some similarities to the database producer.

Consumer: Allows a user or application to pose either a continuous, history, latest-state, or static query over the global schema. Since there is no support for publishing stored relations, there is no need to support queries over such relations. The consumer corresponds exactly with the consumer in the stream integration system above.

The static data published by an on-demand producer is subtly different from the data that would be published by the database producer in the stream integration system presented above. For instance, in the stream integration system it would be possible to relate data in a stream relation with data in a stored relation. This is not possible with the static relations, it is not even possible to relate data in two different

static relations unless they are published by the same on-demand producer. The on-demand producer is included in R-GMA as a mechanism to publish any additional information into the system. It is the responsibility of the user to compute any relationships between the data published by an on-demand producer and any other producer, and to decide upon the semantics of such answers. The on-demand producer is a first step to providing the functionality of the database producer.

At present, to overcome the absence of being able to publish stored data and relate it to the streaming data such stored data is periodically published into the system as a stream. For example, consider the CECluster relation for publishing details of a cluster of computing elements

$$\mathsf{CECluster}(\underline{\mathsf{clusterId}},\ \mathsf{name},\ \mathsf{URL}), \tag{4.6}$$

where clusterId is a unique identifier for a cluster of computing elements, name is the common name for the cluster (which might not be unique), and URL gives the location of the access point for the cluster. While this method of publishing the data periodically does generate excess network traffic, this has not been a problem in current deployments. To develop mechanisms to process stored relations together with stream relations requires additional research that is beyond the scope of this thesis.

4.2.2 Query Answering in R-GMA

The description of the query answering mechanisms in R-GMA can be broken down into two parts. The first is how R-GMA answers a continuous query and the second how it answers a one-off query. (A static query is passed to the on-demand producer that publishes the relation.) The mechanisms will be discussed in the following sections.

Answering a Continuous Query

In R-GMA, a continuous query posed by a consumer consists of a select-project query over a single relation while a continuous query posed by a secondary producer is limited to a selection query over a single relation. Continuous queries can only be answered by primary producers. Since the streams published by the primary producers are disjoint, there is no redundancy in the data. Thus, the query plan used to

answer a continuous query consists of contacting all primary producers which have a view condition that does not contradict the condition in the global query. Each primary producer then streams those tuples that satisfy the query condition directly to the consumer or secondary producer posing the query. Details of how such plans are computed are discussed in Section 7.2.

While this approach to answering continuous queries has proved to be adequate for current deployments of R-GMA, there have been indications that this will not remain the case. This approach requires that a consumer contacts every primary producer that publishes for the named global relation and does not have a view condition that contradicts the query condition. Thus, it must maintain several connections. However, there is a physical limit to the number of connections that each component can maintain. Also, there is a performance cost to maintaining each of these connections. An approach to continuous query answering that exploits the ability of the secondary producers to merge several streams and make the resulting stream available would alleviate this problem. The theory required for such an approach is the subject of this thesis.

Answering a One-Off Query

In R-GMA, the secondary producers provide a mechanism by which one-off queries can be answered. The queries are arbitrary SQL queries, i.e. they can include joins, negation, aggregation, etc. A secondary producer is normally created to collect all of the data appearing on several streams and to maintain either the history of these streams or the latest-state values.

When a one-off query is posed, the registry service identifies those primary and secondary producers which publish all of the relations involved in the query and maintain the appropriate type of data, i.e. either the history of the desired length or latest-state values. It would require complex reasoning to identify only those producers that publish all the relevant data for a query. The consumer posing the query is then given a choice of those secondary producers which publish the entirety of all the relations involved in the query. Where such a secondary producer does not exist, the consumer can choose to contact one of the primary producers but there is no guarantee that it will get the complete answer in this case. Full details of the one-off query answering

mechanism are provided in [108]

4.3 Comparison to Requirements and Existing Systems

This section considers whether the designed stream integration system meets the requirements of a Grid information and monitoring system that were identified in Section 3.2. It will also compare the partial implementation, R-GMA, with existing Grid information and monitoring systems.

4.3.1 Meeting the Requirements

The first requirement identified was the ability to publish both stored and streaming data by advertising the content and being able to answer requests for the data. The proposed stream integration system is designed to allow for the publication of both types of data and, by declaring a view on the global schema, it advertises the content. The producer agents are designed to provide the query answering capabilities. However, the R-GMA implementation of the design only allows for the publication of streaming data at present.

The second requirement was that it should be possible to locate and query data. This requirement is met by the use of a global schema and the fact that consumers pose queries over the schema. The R-GMA implementation provides rudimentary mechanisms to translate continuous and one-off global queries into queries over the available data sources. The rest of this thesis will consider how continuous selection queries can be answered more efficiently by using republishers.

The third requirement states that the system should be scalable, robust, and perform well under the loads anticipated on a large Grid. By separating out the tasks of locating and retrieving data, the stream integration system will be scalable. Scalability and performance will also be achieved by allowing the partial answers generated by republishers to be used in answering any continuous query. The theory to allow this for selection queries is developed in the subsequent chapters of this thesis. The robustness of the system is increased by replicating the registry and

schema services. This has not yet been achieved in the R-GMA implementation. The robustness will also come from updating query plans to reflect changes to the set of publishers in the system. The theory for this will be presented in Chapter 6.

The final requirement was that of security. This is not addressed by the design. R-GMA has a security framework built-in to the implementation. The result is that only producers which a consumer is allowed to access are used to answer a query. Any implementation of the stream integration system would need to consider the security requirements of their application and the effects of these on the query planning process.

4.3.2 Comparison to Other Grid Information and Monitoring Systems

The following considers how R-GMA compares with the existing Grid information and monitoring systems of Section 3.3.3.

R-GMA is a generic unified Grid information and monitoring system. This distinguishes it from Grid monitoring systems such as autopilot [97], GrRADS [98], or the network weather service [9, 100] which have been designed for only specific information.

R-GMA uses the relational data model which means that the schema does not need to be designed with all possible queries in mind. This allows it to answer more complex queries than SCALEA-G [101] or MDS [103, 104].

4.4 Summary

This chapter has proposed an architecture for a stream integration system along with details of the R-GMA system, a partial implementation. The stream integration system presents a virtual dataspace via a global schema. The design discussed mechanisms for publishing and query the data in the virtual dataspace. A key feature of the architecture is the republisher. Not only does the republisher allow history and latest-state queries to be answered, it also provides an infrastructure for answering continuous queries more efficiently. The theory required to exploit the republishers for answering

continuous selection queries will be presented in the next two chapters.

Chapter 5

Answering Continuous Queries Using Views

This chapter considers the problem of answering a continuous query over the global schema using the data published by the available publishers, i.e. it considers how to integrate streams of data and generate a query plan that efficiently answers a global query from the available data sources. A solution is proposed that is based on answering queries using views over data streams. This is needed to allow a stream integration system, such as that proposed in Chapter 4, to answer queries and to provide guarantees about the answer streams generated. The components of the stream integration system will be referred to throughout this chapter. The theory developed is for selection queries which provides many challenges, and matches the requirements of the motivating Grid information and monitoring system problem. The theory presented is general and can be applied whenever there is a need to integrate distributed streams of data and has been published in [109].

The chapter begins by presenting a formalisation of the problem in Section 5.1. It presents a theoretical data model for a data stream and defines when a data stream conforms to a schema as well as providing basic operations and properties for a stream. It then defines how a stream is published by a producer and how it can be queried with a continuous selection query.

Section 5.2 introduces a formalisation of republishers. Since republishers introduce redundancy in the data, a query plan must decide from where to retrieve each part of the answer stream. Thus, desirable properties for a query plan are identified and

formalised.

Finally, Section 5.3 presents a mechanism for generating query plans for a continuous global selection query that exploits the redundancy introduced in the data by the republishers. It will be shown that the plans generated guarantee the desirable properties for a query plan. The same theory and mechanism can be used to answer continuous select-project queries.

5.1 A Formal Framework for Publishing and Querying Data Streams

5.1.1 A Global Schema

In order for the components of the framework to be able to communicate, it is assumed that there is a *global schema* against which queries are posed, as stated in Section 4.1.3. A relation r in the global schema consists of attributes with defined types. The attributes of r are split into three parts: key attributes, measurement attributes, and a timestamp attribute.

As an example, taken from a Grid information and monitoring application, consider the relation ntp ("network throughput") as introduced earlier (4.1) with the schema

$$\mathsf{ntp}(\underline{\mathsf{from}}, \underline{\mathsf{to}}, \underline{\mathsf{tool}}, \underline{\mathsf{psize}}, \mathsf{latency}, [\mathsf{timestamp}]), \tag{5.1}$$

which records the time it took, according to some particular tool, to transport packets of a specific size from one node to another. Again, the types are omitted for clarity of presentation. The underlined attributes make up the primary key of ntp, while latency is the measurement attribute, and timestamp records the time at which the reading was made.

5.1.2 Streams and Their Properties

Data streams are modelled as finite or infinite sequences of tuples where one attribute of each tuple is a timestamp. This captures the idea that a stream consists of readings, each of which is taken at a specific point in time.

A stream s satisfies the schema of a relation r if all its tuples satisfy the description of r. It is assumed that a relation schema is declared for every stream and that the stream satisfies its *local* schema.

More precisely, suppose that T is the set of all tuples that can be derived for the relation r. Then a stream s that conforms to the relation r is a partial function from the natural numbers $\mathbb{N}$ to T,

$$s\colon \mathbb{N} \hookrightarrow T, \tag{5.2}$$

such that, for some $m, n \in \mathbb{N}$, if $s(n)$ is defined then the tuple $s(m)$ is defined for all $m < n$. Thus, $s(n)$ denotes the n^{th} tuple of s. The notation $s(n) = \perp$ is used if the n^{th} tuple of s is undefined. A special case is the empty stream, also denoted as $\perp$, which is undefined for every $n \in \mathbb{N}$.

This choice of model for data streams allows different tuples to have the same timestamp, e.g. if the stream is created by merging several other streams, and tuples to arrive in an order independent of their timestamp. Thus, there are no requirements about how regularly a reading can be taken nor is it required that readings are published in chronological order.

Properties of Data Streams

As stated in Section 5.1.1 the attributes of a stream relation are split into three parts: key attributes, measurement attributes and the timestamp. The following shorthand notations will be used for the subtuples of $s(n)$ relating to these three parts:

$s^{\kappa}(n)$ for the values of the key attributes;
$s^{\mu}(n)$ for the values of the measurement attributes;
$s^{\tau}(n)$ for the timestamp of $s(n)$.

This notation is used to formalise the channels of a stream which were informally introduced in Section 4.1.3. A stream s_1 is a *substream* of s_2 if s_1 can be obtained from s_2 by deleting zero or more tuples from s_2. A *channel* of s is a maximal substream whose tuples agree on the key attributes of s, i.e. for any $s(n)$ and $s(m)$ that occur on the same channel then $s^{\kappa}(n) = s^{\kappa}(m)$.

The following properties of streams are central to the semantics of stream queries considered here.

Duplicate Freeness: A stream s is *duplicate free* if for all m, n with $m \neq n$ it is the case that $s(m) \neq s(n)$, that is, if no tuple occurs twice in s.

Weak Order: A stream s is *weakly ordered* if for all m, n with $s^\kappa(m) = s^\kappa(n)$ and $m < n$ it is the case that $s^\tau(m) < s^\tau(n)$. This means that in every channel of s, tuples appear in the order of their timestamps. Note that this definition is equivalent to requiring that for all m, n with $s^\kappa(m) = s^\kappa(n)$ and $s^\tau(m) < s^\tau(n)$ it is the case that $m < n$.

Disjointness: Two streams s_1 and s_2 are *disjoint* if for all m, n we have that $s_1(m) \neq s_2(n)$, that is, if s_1 and s_2 have no tuples in common.

Channel Disjointness: Two streams s_1 and s_2 are *channel disjoint* if for all m, n it is the case that $s_1^\kappa(m) \neq s_2^\kappa(n)$, that is, if s_1 and s_2 have no channels in common.

Clearly, channel disjointness implies disjointness.

Operations on Streams

The following contains two simple definitions for operations on streams. Let s be a stream and suppose that C is a condition involving attributes of the schema of s, constants, operators "$=$", "$\neq$", "$<$", "$\leq$", "$>$", "$\geq$", and boolean connectives. Then the *selection* $\sigma_C(s)$ of s is the substream of s that consists of the tuples in s that satisfy C where those tuples appear in the same order as they do in s.

Let $s_1, \ldots, s_n$ be streams for relations with union compatible schemas. A stream s is a *union* of $s_1, \ldots, s_n$ if s can be obtained by merging these streams, i.e. if each s_i contributes all its tuples to s, and the tuples of s_i occur in s in the same order as they do in s_i.

The result of a selection is unique while this is not the case for a union. This is because the union operation does not uniquely define the order when merging two streams. Also note that

1. The stream resulting from a selection operation is weakly ordered if its argument stream is,

2. The streams that can result from the union operation are weakly ordered if the argument streams are channel disjoint and weakly ordered, and

3. The result of a union is duplicate free if the argument streams are mutually disjoint and duplicate free.

5.1.3 Producing a Data Stream

A *stream producer* is a component that is capable of producing a data stream. Every stream producer has a local relation schema. Both the stream producer and the name of the relation in the local schema for that producer are denoted by the letter S.

Local Queries over Stream Producers

Queries posed over the stream producers are called *local queries* as opposed to *global queries*, which are posed over a global schema and shall be discussed in Section 5.1.4.

The local queries considered are unions of selections of the form

$$Q = \sigma_{C_1}(S_1) \uplus \ldots \uplus \sigma_{C_m}(S_m), \tag{5.3}$$

where $S_1, \ldots, S_m$ are distinct stream producers whose schemas are union compatible. A special case is the empty union, which results in the empty stream ε.

To define the semantics of such a query, a stream is associated with each producer. A *stream assignment* over a set of producers is a mapping $\mathcal{I}$ that associates with each producer S a stream $S^{\mathcal{I}}$ that is compatible with the schema of S. A stream s is an *answer* for Q w.r.t. $\mathcal{I}$ if s is a multi-set union of the selections $\sigma_{C_1}(S_1^{\mathcal{I}}), \ldots, \sigma_{C_m}(S_m^{\mathcal{I}})$. It should be noted that an answer is not uniquely defined since there is more than one way to merge the selections $\sigma_{C_i}(S_i^{\mathcal{I}})$. The empty union ε has only one answer, namely the empty stream $\perp$.

The multi-set, or bag, union is used as duplicate elimination would be a costly operation and not realistic as it would result in a performance burden. In the worst case, where the stream is infinite, it would require an infinite amount of storage to ensure that all duplicates were eliminated. Although the difference between multi-set union and set union does not make a difference when only producers are considered, there will be a difference when republishers are considered.

Producer Configurations

Consider the collections of stream producers as they would be created in an implementation of the stream integration system. It is assumed that there is a *global schema* $\mathcal{G}$, which is a collection of stream relations. A *producer configuration* consists of a finite set $\mathcal{S}$ of stream producers and a mapping v that associates with each producer $S \in \mathcal{S}$ a query v_S over the global schema $\mathcal{G}$ such that v_S is compatible with the schema of S. If no confusion can arise, the producer configuration is also denoted with the letter $\mathcal{S}$. In the stream integration system, producer configurations are represented in the schema and the registry.

The query v_S is called the *descriptive view* of the producer S. Descriptive views are limited to selections, i.e. they have the form $\sigma_D(r)$ where D is a condition and r is a global relation. Although only the simple case of selection queries are considered here, this already presents significant challenges. Moreover, developing the framework for selection queries meets the requirements currently identified for a Grid information and monitoring system which was the motivating application for this work.

To ease the presentation, it is required that if S is described by the view $\sigma_D(r)$ that S and r have the same attributes with the same types and the same key constraints. It is also required that the condition D in $\sigma_D(r)$ involves *only key attributes* of r. Thus, the view of a producer restricts the channels, but not the possible measurements of the readings.

Instances of Producer Configurations

A producer configuration is similar to a database schema. It contains declarations and constraints, but no data. The following shall define which streams are the possible instances of such a configuration.

A stream s is *sound* with respect to a query $\sigma_D(r)$ over the global schema if the schema of s is compatible with the schema of r and if every tuple $s(n)$ satisfies the view condition D.

An assignment $\mathcal{I}$ for the producers in a configuration $\mathcal{S}$ is an *instance* of $\mathcal{S}$ if for every $S \in \mathcal{S}$ the following all hold for the stream $S^{\mathcal{I}}$:

1. Sound with respect to the descriptive view $v(S)$.

2. Duplicate free.

3. Weakly ordered.

4. Channel disjoint from the streams of the other producers.

5.1.4 Global Queries and Query Plans

Consumer components in R-GMA pose queries over the global schema and receive a stream of answers. The only queries over the global schema considered in this thesis are selections of the form

$$q = \sigma_C(r), \tag{5.4}$$

where r is a global relation. Since the relation r does not refer to an existing stream it is not straightforward what the answer to such a query should be.

Intuitively, the query q is posed against a virtual stream, made up of all the individual streams contributed by the producers. A producer S *produces for* the relation r if S is described by a view over r. If $\mathcal{I}$ is a producer instance then an *answer* for q w.r.t. $\mathcal{I}$ is a duplicate free and weakly ordered stream that consists of those tuples satisfying C that occur in streams $S^{\mathcal{I}}$ of producers S that produce for r. Note that according to the definition there can be infinitely many different answer streams for a query q. Any two answer streams consist of the same tuples but differ in the order in which they appear.

Note also that tuples do not need to occur in the same order as in the original producer streams. It is only required that the tuples of a channel appear in the same order as in the stream of the publishing stream producer. This makes it possible for streams to be split and then re-merged during processing.

Since global queries cannot be answered directly, they need to be translated into local queries. A local query Q is a *plan* for a global query q if for every producer instance $\mathcal{I}$ it is the case that all answer streams for Q w.r.t. $\mathcal{I}$ are also answer streams for q w.r.t. $\mathcal{I}$. Proposition 5.1 gives a characterisation of plans that use only stream producers.

Proposition 5.1 (Plans Using Producers) *Let* $Q = \sigma_{C_1}(S_1) \uplus \ldots \uplus \sigma_{C_m}(S_m)$ *be a local query where each* S_i *is described by a view* $\sigma_{D_i}(r)$ *and let* $q = \sigma_C(r)$ *be a global*

query. Then Q is a plan for q if and only if the following both hold.

1. *For each $i \in 1..m$ it must be the case that*

$$C_i \wedge D_i \models C \qquad \text{and} \qquad C \wedge D_i \models C_i. \tag{5.5}$$

2. *Every stream producer S with a descriptive view $\sigma_D(r)$ such that $C \wedge D$ is satisfiable occurs as some S_i.*

Proof. The first condition ensures that any producer occurring in Q contributes only tuples satisfying the query and that it contributes all such tuples that it can possibly produce. The second condition ensures that any producer that can possibly contribute occurs in the plan.

Since by assumption all S_i are distinct and the streams of distinct producers are channel disjoint, all answers of Q are duplicate free. Also, by the definition of union of streams, all answers are weakly ordered. □

5.2 Query Plans Using Republishers

This section formally introduces republishers and generalises query plans accordingly. Characteristic criteria are developed that can be used to check whether a local query over arbitrary publishers is a plan for a global query.

5.2.1 Republishers and Queries over Republishers

A *republisher* R is a component that is defined by a global query $q_R = \sigma_D(r)$. For a given instance $\mathcal{I}$ of a producer configuration the republisher outputs a stream that is an answer to q_R w.r.t. $\mathcal{I}$. The descriptive view $v(R)$ of a republisher is identical to the defining query q_R. A *republisher configuration* $\mathcal{R}$ is a set of republishers.

Publisher Configurations

Since both producers and republishers publish streams, they are collectively referred to as *publishers*. Ultimately, the aim is to answer global queries using arbitrary publishers.

A *publisher configuration* is defined as a pair $\mathcal{P} = (\mathcal{S}, \mathcal{R})$ consisting of a producer and a republisher configuration. By abuse of notation, the set $\mathcal{S} \cup \mathcal{R}$ shall be identified as $\mathcal{P}$.

A stream assignment $\mathcal{J}$ for publishers in $\mathcal{P}$ is an *instance* of $\mathcal{P}$ if

1. The restriction $\mathcal{J}_{|\mathcal{S}}$ of $\mathcal{J}$ to $\mathcal{S}$ is an instance of $\mathcal{S}$, and if
2. For every republisher R the stream $R^{\mathcal{J}}$ is an answer for the global query q_R w.r.t. $\mathcal{J}_{|\mathcal{S}}$.

Thus, an instance $\mathcal{J}$ is "essentially" determined by $\mathcal{J}_{|\mathcal{S}}$. Note that $R^{\mathcal{J}}$ being an answer for a global query implies that $R^{\mathcal{J}}$ is duplicate free and weakly ordered.

Local Queries over Publishers

In the presence of republishers the concept of a local query, which had the form (5.3), is generalised in such a way as to allow them to be posed over arbitrary publishers. Thus, general local queries have the form

$$Q = \sigma_{C_1}(P_1) \uplus \ldots \uplus \sigma_{C_m}(P_m), \tag{5.6}$$

where $P_1, \ldots, P_m$ are distinct publishers.

A stream s is an *answer* for Q w.r.t. $\mathcal{J}$ if s is a union of the selections

$$\sigma_{C_1}(P_1^{\mathcal{J}}), \ldots, \sigma_{C_m}(P_m^{\mathcal{J}}). \tag{5.7}$$

Similarly as before, a local query Q as in (5.6) is a *plan* for a global query q if for all instances $\mathcal{J}$, every answer for Q is an answer for q.

Republishers add to the difficulty of characterising when a local query over a publisher configuration is a plan for a global query because they introduce redundancy. As a consequence, answers to such a query need not be duplicate free or weakly ordered.

5.2.2 Properties of Query Plans

The characteristic properties of plans are identified here. They are defined in terms of the properties of the answers to a query.

Consider a fixed publisher configuration $\mathcal{P}$ and let Q be a query over $\mathcal{P}$ as in (5.6). Then Q is *duplicate free* if for all instances $\mathcal{J}$ of $\mathcal{P}$ all answer streams for Q w.r.t. $\mathcal{J}$ are duplicate free. In a similar way, query Q is defined to be *weakly ordered.* Let q be a global query. Then Q is *sound* for q if for all instances $\mathcal{J}$ of $\mathcal{P}$ all answer streams for Q w.r.t. $\mathcal{J}$ are sound for q. A stream s is *complete* for q with respect to a producer instance $\mathcal{I}$ if every tuple in an answer stream for q w.r.t. $\mathcal{I}$ occurs also in s. Query Q is *complete* for q if for all instances $\mathcal{J}$ all answer streams for Q w.r.t. $\mathcal{J}$ are complete for q w.r.t. $\mathcal{J}_{|\mathcal{S}}$.

Clearly Q is a plan for q if and only if the following all hold for Q:

1. Sound for q.

2. Complete for q.

3. Duplicate free.

4. Weakly ordered.

For soundness and completeness it would be expected that there are characterisations similar to those in Proposition 5.1. However, with republishers there is the difficulty that the descriptive views do not accurately describe which data a republisher offers in a given configuration. For instance, a republisher may always publish the empty stream if the configuration does not contain any producers whose views are compatible with the republisher's query.

Given a publisher configuration $\mathcal{P}$, for every republisher R defined by the query $\sigma_D(r)$, a new condition D' is derived as follows. Let $S_1, \ldots, S_n$ be all producers for r in $\mathcal{P}$, where $v(S_i) = \sigma_{E_i}(r)$. Then

$$D' = D \ \wedge \ (\bigvee_{i=1}^{n} E_i). \tag{5.8}$$

Intuitively, D' describes which of the tuples that can actually be produced in $\mathcal{P}$ will be republished by $\mathcal{R}$. The view $v'(R) := \sigma_{D'}(r)$ is called the *relativisation* of the view $v(R)$ w.r.t. $\mathcal{P}$. For a producer S the relativisation $v'(S)$ is defined to be equal to $v(S)$. Note that an empty disjunction is equivalent to *false* and therefore the relativised condition for a republisher that does not have producers is *false*.

5.2.3 Soundness

A characterisation of soundness is given in Theorem 5.2.

Theorem 5.2 (Soundness of a query plan) *Let $\mathcal{P}$ be a publisher configuration, $q = \sigma_C(r)$ a global query, and $Q = \sigma_{C_1}(P_1) \uplus \cdots \uplus \sigma_{C_m}(P_m)$ be a local query over $\mathcal{P}$. Suppose that the descriptive view of P_i is $v(P_i) = \sigma_{D_i}(r)$ and that the relativisation is $v'(P_i) = \sigma_{D'_i}(r)$. Then Q is* sound for q *if and only if for each $i \in 1..m$ it is the case that*

$$C_i \ \wedge \ D'_i \models C. \tag{5.9}$$

Proof. Clearly, if (5.9) holds, then every tuple in an answer to $\sigma_{C_i}(r)$ over $\mathcal{P}$ satisfies C, and so does every tuple in an answer to Q over $\mathcal{P}$.

Conversely, if (5.9) does not hold, then there is a tuple t that satisfies some C_i and D'_i, but not C. Since the argument is simpler if P_i is a producer, it is assumed without loss of generality that P_i is a republisher.

Since t satisfies D'_i, there is a producer S with $v(S) = \sigma_E(r)$ such that t satisfies D_i and E. Let $\mathcal{J}$ be an instance where the stream $S^{\mathcal{J}}$ contains t. Then the stream $P_i^{\mathcal{J}}$ contains t as well, because $P_i^{\mathcal{J}}$ is an answer for $\sigma_{D_i}(r)$. Then t is in every answer stream for $\sigma_{C_i}(P_i)$ and therefore in every answer stream for Q w.r.t. $\mathcal{J}$. However, t does not occur in any answer stream for Q because t does not satisfy C. □

It is easy to see that the criterion of Theorem 5.2 can be weakened to a sufficient one if instead of (5.9) it is required that for each $i \in 1..m$ it is the case that

$$C_i \ \wedge \ D_i \models C, \tag{5.10}$$

where D_i is the original condition in the descriptive view of P_i.

5.2.4 Completeness

To characterise completeness, it must be possible to distinguish between the producers and the republishers in a local query. The reason is that the stream of a republisher is always complete for its descriptive view while this need not be the case for a producer.

Let Q be a local query as in (5.6) and suppose that $R_1 \ldots, R_k$ are the republishers and $S_1 \ldots, S_l$ the stream producers among $P_1, \ldots, P_m$. Then the query can be written

as $Q = Q^{\mathbf{R}} \uplus Q^{\mathbf{S}}$ where

$$Q^{\mathbf{R}} = \sigma_{C_1}(R_1) \uplus \cdots \uplus \sigma_{C_k}(R_k) \tag{5.11}$$

$$Q^{\mathbf{S}} = \sigma_{C'_1}(S_1) \uplus \cdots \uplus \sigma_{C'_l}(S_l). \tag{5.12}$$

Suppose that the republishers have the descriptive views $v(R_i) = \sigma_{D_i}(r)$. Then a condition $C_Q^{\mathbf{R}}$, which summarises the conditions in the selections of the republisher part $Q^{\mathbf{R}}$ of Q, is defined as follows:

$$C_Q^{\mathbf{R}} = \bigvee_{j=1}^{k} (C_j \ \wedge \ D_j). \tag{5.13}$$

A characterisation for completeness is now given in Theorem 5.3.

Theorem 5.3 (Completeness of a query plan) *Let $\mathcal{P}$ be a publisher configuration, $q = \sigma_C(r)$ a global query, and $Q = Q^{\mathbf{R}} \uplus Q^{\mathbf{S}}$ a local query where $Q^{\mathbf{R}}$ and $Q^{\mathbf{S}}$ are as in* (5.11) *and* (5.12)*. Then Q is* complete *for q if and only if for every stream producer $S \in \mathcal{P}$, where S is described by the view $\sigma_E(r)$, one of the two following statements hold.*

1. *$S = S_i$ for some producer S_i in $Q^{\mathbf{S}}$ and*

$$C \ \wedge \ E \models C_Q^{\mathbf{R}} \vee C'_i; \tag{5.14}$$

2. *S does not occur in $Q^{\mathbf{S}}$ and*

$$C \ \wedge \ E \models C_Q^{\mathbf{R}}. \tag{5.15}$$

Proof. (*Sketch*) To see that the criterion is sufficient note that any tuple in an answer for q must satisfy C and must originate from some producer for r with view condition E. Let S be such a producer. A tuple returned by Q can occur either as an element of an answer for $Q^{\mathbf{R}}$ or as an element of an answer for $Q^{\mathbf{S}}$. If S is present in Q. then (5.14) guarantees that a tuple produced by S is either returned by $Q^{\mathbf{R}}$ or by $Q^{\mathbf{S}}$. If S is not present in Q, then (5.15) guarantees that a tuple produced by S is returned by $Q^{\mathbf{R}}$.

To see that the criterion is necessary, assume that there is producer S for which neither of the two statements holds. Suppose that S occurs in $Q^{\mathbf{S}}$. Then there is a

tuple t such that t satisfies $C \wedge E$, but satisfies neither $C_Q^{\mathbf{R}}$ nor C_i'. That is, there exists an instance $\mathcal{J}$ of $\mathcal{P}$ such that t occurs in the stream $S^{\mathcal{J}}$. Every answer for q w.r.t. $\mathcal{J}$ contains t. However, t does not occur in any answer for Q w.r.t. $\mathcal{J}$. With a similar argument one can show that t does not occur in any answer for Q if S does not occur in $Q^{\mathbf{S}}$. In summary, this proves that Q is not complete for q. □

5.2.5 Duplicate Freeness

A characterisation of duplicate freeness is provided in Theorem 5.4.

Theorem 5.4 (Duplicate Freeness of a query plan) *Suppose $\mathcal{P}$ is a publisher configuration and Q a local union query over publishers $P_1, \ldots, P_m$ as in* (5.6)*. Suppose that the relativised descriptive view of each P_i is $v'(P_i) = \sigma_{D_i'}(r)$. Then Q is* duplicate free *if and only if the condition*

$$(C_i \wedge D_i') \wedge (C_j \wedge D_j') \tag{5.16}$$

is unsatisfiable for each republisher P_i and publisher P_j where $i \neq j$.

Proof. If the statement is true, then for any instance $\mathcal{J}$, the streams $\sigma_{C_i}(P_i^{\mathcal{J}})$ are mutually disjoint and every answer of Q is duplicate free because the streams $\sigma_{C_i}(P_i^{\mathcal{J}})$ are duplicate free.

If the statement is not true, then there are i and j with $i \neq j$ and a tuple t such that t satisfies both $C_i \wedge D_i'$ and $C_j \wedge D_j'$. Suppose that P_i is a republisher and P_j is a producer. Consider an instance $\mathcal{J}$ where t occurs in the stream $P_j^{\mathcal{J}}$ of the producer P_j. Since P_i is a republisher, t occurs also in the stream $P_i^{\mathcal{J}}$. Finally, since t satisfies both C_i and C_j, the tuple occurs in both streams, $\sigma_{C_i}(P_i^{\mathcal{J}})$ and $\sigma_{C_j}(P_j^{\mathcal{J}})$. Hence, there is an answer to Q where the tuple t occurs twice.

If both P_i and P_j are republishers, it can shown that there is a producer S with view $\sigma_E(r)$ such that $D_i \wedge D_j \wedge E$ is satisfiable. Then choose a satisfying tuple t and consider an instance $\mathcal{J}$ where $S^{\mathcal{J}}$ contains t. The rest of the argument is analogous to the first case. □

Similar to Theorem 5.2, the criterion of the above theorem can be turned into a sufficient one by replacing the relativised conditions D_i' in (5.16) by the view conditions

D_i, that is, if

$$(C_i \wedge D_i) \wedge (C_j \wedge D_j) \tag{5.17}$$

is unsatisfiable for each republisher P_i and publisher P_j where $i \neq j$.

5.2.6 Weak Order

The following lemma gives a semantic characterisation of weakly ordered queries.

Lemma 5.5 *Let $\mathcal{P}$ be a publisher configuration and $Q = \sigma_{C_1}(P_1) \uplus \cdots \uplus \sigma_{C_m}(P_m)$ be a local query. Then Q is weakly ordered if and only if for all publishers P_i, P_j with $i \neq j$ occurring in Q and for every instance $\mathcal{J}$ of $\mathcal{P}$ the following holds:*

If t and t' are tuples occurring in the two streams $\sigma_{C_i}(P_i^{\mathcal{J}})$ and $\sigma_{C_j}(P_j^{\mathcal{J}})$, respectively, then t and t' disagree on their key attributes.

The lemma holds because otherwise the two streams in question could be merged in such a way that t and t' occur in an order that disagrees with their timestamps. The lemma excludes, for instance, the possibility to use two republishers $R_{>10}$ and $R_{\leq 10}$ with views $\sigma_{\mathsf{latency}>10}(\mathsf{ntp})$ and $\sigma_{\mathsf{latency}\leq 10}(\mathsf{ntp})$, respectively, for answering the query $\sigma_{true}(\mathsf{ntp})$. The reason is that, latency being a measurement attribute, some tuples of a given channel could end up being republished by $R_{>10}$ and others by $R_{\leq 10}$.

Since in the end the goal is to characterise plans for global queries, the following considers when a local query is weakly ordered *and* complete for some global query q. Considering these two properties together has the advantage that it leads to a characterisation in terms of the individual disjuncts that make up a union query.

Lemma 5.6 (One Publisher per Channel) *Let $\mathcal{P}$ be a publisher configuration and $Q = \sigma_{C_1}(P_1) \uplus \cdots \uplus \sigma_{C_m}(P_m)$ be a local query. Suppose that Q is complete for the global query $\sigma_C(r)$. Then Q is weakly ordered if and only if for every publisher P_i occurring in Q and every instance $\mathcal{J}$ of $\mathcal{P}$ the following holds:*

If the stream $\sigma_{C_i}(P_i^{\mathcal{J}})$ contains some *tuple t that satisfies C, then this stream contains* every *tuple t' that is generated by a producer for r such that t' satisfies C and t' agrees with t on the key attributes.*

This lemma follows immediately from the preceding one: if it is impossible for two publishers to publish tuples from the same channel, then all tuples of one channel must come from the same publisher.

Lemma 5.6 can be formalised in logic. The condition C of query q is written as $C(x, y)$, where x stands for the vector of key attributes of r, which identifies a channel, and y for the non-key attributes. Similarly, the conditions C_i in query Q and D'_i in the relativised descriptive views are written as $C_i(x, y)$ and $D'_i(x, y)$ and the conjunction $C_i(x, y) \ \wedge \ D'_i(x, y)$ is abbreviated as $F_i(x, y)$.

Theorem 5.7 (Weak Order) *Let $\mathcal{P}$ be a publisher configuration, Q a local query over $\mathcal{P}$, where $Q^{\mathbf{R}} = \sigma_{C_1}(R_1) \uplus \cdots \uplus \sigma_{C_k}(R_k)$, and $q = \sigma_C(r)$ a global query. Suppose that Q is complete for q w.r.t. $\mathcal{P}$. Then Q is* weakly ordered *if and only if for all $i \in 1..k$ it is the case that*

$$\exists y.\, (C(x, y) \ \wedge \ F_i(x, y)) \models \forall y.\, (C(x, y) \ \rightarrow \ F_i(x, y)). \tag{5.18}$$

Proof. Suppose that (5.18) holds for all $i \in 1..k$. Consider an instance $\mathcal{J}$ of $\mathcal{P}$. Then the claim can be shown using Lemma 5.6.

Suppose that $t = (t_x, t_y)$ is a tuple in the stream $\sigma_{C_i}(R_i^{\mathcal{J}})$ obtained from a republisher R_i. Then t_x satisfies $\exists y.\, (C(x, y) \ \wedge \ F_i(x, y))$. By (5.18), it follows that t_x also satisfies $\forall y.\, (C(x, y) \ \rightarrow \ F_i(x, y))$. Let $t' = (t_x, t'_y)$ be a tuple that is generated by a producer for r and agrees with t on the key attributes. Suppose that t' satisfies C. Then, since t_x satisfies $\forall y.\, (C(x, y) \ \rightarrow \ F_i(x, y))$, it follows that t' also satisfies F_i. Hence, t' occurs also in the stream $\sigma_{C_i}(R_i^{\mathcal{J}})$.

Since producer streams do not share channels, Lemma 5.6 yields the sufficiency of the criterion.

Next the necessity is shown. Suppose that (5.18) does not hold for some $i \in 1..k$. Then there is a tuple $t = (t_x, t_y)$ that satisfies $C \ \wedge \ F_i$ and a tuple $t' = (t_x, t'_y)$ such that t' satisfies C, but not F_i. By definition of F_i, the tuple t satisfies C_i, D_i, and some condition E for a stream producer S with descriptive view $\sigma_E(r)$. An instance $\mathcal{J}$ can be constructed where both t and t' occur in the stream of S. Then t occurs in every answer to $\sigma_{D_i}(r)$, the defining query of R_i, and thus in $R_i^{\mathcal{J}}$. Moreover, t occurs in the stream $\sigma_{C_i}(R_i^{\mathcal{J}})$. However, since t' does not satisfy F_i, it does not occur in that stream. Hence, by Lemma 5.6 it follows that Q is not weakly ordered. □

It is noted that the proof above would go through as well if (5.18) were changed into

$$\exists y.\,(C(x,y)\ \wedge\ C_i(x,y)\ \wedge\ D'_i(x,y)) \models \forall y.\,(C(x,y) \rightarrow C_i(x,y)\ \wedge\ D_i(x,y)). \quad (5.19)$$

where the relativised condition D'_i on the right hand side of the entailment in (5.18) is replace by the original view condition D_i. However, this formulation is less concise.

Consider again that part of the proof above that shows the sufficiency of the fact that (5.18) holds for all $i \in 1..k$ for the claim of Theorem 5.7. It turns out that the proof also works for the definition

$$F_i(x,y) = C_i(x,y)\ \wedge\ D_i(x,y), \quad (5.20)$$

that is, if the relativised view is replaced by original view conditions. Thus, (5.20) leads to a simpler albeit sufficient criterion for weak order.

The entailment in (5.18) of Theorem 5.7 is in general difficult to check because of the universal quantifier. However, it can be simplified for queries and descriptive views where the conditions on key and on non-key attributes are decoupled, that is, if every condition $C(x,y)$ can be written equivalently as $C^\kappa(x)\ \wedge\ C^\mu(y)$ (and analogously C_i and D_i, and therefore also F_i). This restriction is not likely to cause difficulties in practice.

Theorem 5.8 *Suppose* $C(x,y) \equiv C^\kappa(x)\ \wedge\ C^\mu(y)$ *and* $F_i(x,y) \equiv F_i^\kappa(x)\ \wedge\ F_i^\mu(y)$. *Then*

$$\exists y.\,(C(x,y)\ \wedge\ F_i(x,y)) \models \forall y.\,(C(x,y) \rightarrow F_i(x,y)) \quad (5.21)$$

holds if and only if one of the following holds:

1. $C^\kappa(x)\ \wedge\ F_i^\kappa(x)$ *is unsatisfiable.*

2. $C^\mu(y)\ \wedge\ F_i^\mu(y)$ *is unsatisfiable.*

3. $C^\mu(y) \models F_i^\mu(y)$.

Proof. Suppose that (5.21) holds. Let $t(x,y)$ be a tuple for which $\exists y.\,(C(x,y)\ \wedge\ F_i(x,y))$ is true then it is the case that $C^\kappa(x) \wedge F_i^\kappa(x)$ is satisfiable and $C^\mu(y) \wedge F_i^\mu(y)$ is satisfiable. Thus, it is required to show that $C^\mu(y) \models F_i^\mu(y)$. Since (5.21) holds, for

t it is the case that $\forall y.\,(C(x,y) \rightarrow F_i(x,y))$ and by decoupling the conditions that $C^\mu(y) \models F_i^\mu(y)$ holds.

Now it is required to show that $\exists y.\,(C(x,y) \wedge F_i(x,y)) \models \forall y.\,(C(x,y) \rightarrow F_i(x,y))$ holds only if one of the cases holds. Suppose that either case 1 or 2 holds, then the entailment holds since the condition $\exists y.\,(C(x,y) \wedge F_i(x,y))$ is always false.

Suppose that $C^\mu(y) \models F_i^\mu(y)$ holds and that $C^\kappa(x) \wedge F_i^\kappa(x)$ is satisfiable and $C^\mu(y) \wedge F_i^\mu(y)$ is satisfiable. From the fact that $C^\kappa(x) \wedge F_i^\kappa(x)$ is satisfiable and $C^\mu(y) \wedge F_i^\mu(y)$ is satisfiable it follows that $\exists y.\,(C(x,y) \wedge F_i(x,y))$ is satisfiable. Thus, it must be shown that whenever $\exists y.\,(C(x,y) \wedge F_i(x,y))$ holds that $\forall y.\,(C(x,y) \rightarrow F_i(x,y))$ holds. Since $C^\mu(y) \models F_i^\mu(y)$ it follows that the entailment (5.21) holds. □

Again, a sufficient criterion is obtained if in the definition of the F_i the relativised view conditions are replaced by the original ones.

The theorems in this section contain characterisations that can be used to verify whether a local query is a plan for a global query. It has been shown that the characterisations can be simplified to yield sufficient criteria for soundness, duplicate freeness and weak order.

The next section discusses how the characterisations can be used to compute query plans over a publisher configuration. In the next chapter it will be shown how these techniques can be used to realise hierarchies of publishers where republishers consume from other republishers.

5.3 Computing Consumer Query Plans

Based on the characterisations in the previous section, there is a straightforward approach to constructing a plan Q for a global query $q = \sigma_C(r)$. If $S_1, \ldots, S_n$ is a sequence comprising all stream producers in a configuration $\mathcal{P}$ that publish for relation r, then by Proposition 5.1 the query

$$\sigma_C(S_1) \uplus \cdots \uplus \sigma_C(S_n) \tag{5.22}$$

is a plan for q. However, this plan may access a higher number of publishers than necessary because it does not make use of republishers. The question arises when a publisher is potentially useful for a query.

General Assumption. *It is assumed from now on that in global queries and descriptive views the conditions on key and non-key attributes are decoupled, that is, every condition C can be equivalently rewritten as $C^\kappa \wedge C^\mu$, where C^κ involves only key attributes and C^μ involves only non-key attributes.*

5.3.1 Relevant Publishers

This subsection considers which publishers can potentially contribute to a query plan.

A publisher P is *strictly relevant* for a query q with respect to a configuration $\mathcal{P}$ if there is a plan Q for q that contains a disjunct $\sigma_{C'}(P)$ such that for some instance $\mathcal{J}$ of $\mathcal{P}$ the stream $\sigma_{C'}(P^{\mathcal{J}})$ is non-empty.

Proposition 5.9 (Strict Relevance) *Let $\mathcal{P}$ be a publisher configuration and P a publisher with view $\sigma_D(r)$, where $D = D^\kappa \wedge D^\mu$, and where D' is the relativised view condition. Let $q = \sigma_C(r)$ be a global query where $C = C^\kappa \wedge C^\mu$. Then P is* strictly relevant *for q w.r.t. $\mathcal{P}$ if and only if both the following hold.*

1. *$C \wedge D'$ is satisfiable.*
2. *$C^\mu \models D^\mu$.*

Proof. If P is strictly relevant, then criterion 1 holds because P contributes some tuple to q and criterion 2 holds by Theorem 5.7 because the plan containing P is complete and weakly ordered.

Conversely, suppose that the two criteria hold. If P is a producer an instance can be constructed where P produces a tuple satisfying C. Then P can be part of a plan as in (5.22). Because of criterion 1, there is an instance where P contributes at least one tuple to the answer of the plan.

If P is a republisher, by considering the query $Q = \sigma_C(P) \uplus \sigma_{C'}(S_1) \uplus \cdots \uplus \sigma_{C'}(S_n)$, where $S_1, \ldots, S_n$ are all producers for r in $\mathcal{P}$ and $C' = C \wedge \neg D$. Then it is easy to check that Q is duplicate free, and sound and complete for q. Moreover, because of criterion 2, Q is weakly ordered. Finally, criterion 1 allows us to construct an instance of $\mathcal{P}$ where P actually contributes to Q. □

Criterion 1 of Proposition 5.9 involves relativised views. In practice, this is hard to check because there may be a large number of producers in a configuration and

producers may come and go. The criterion can be generalised in such a way that it depends solely on the publisher and the query, Lemma 5.10. Intuitively, the first property states that P can potentially contribute values for *some* channels requested by q, while the second states that for those channels *all* measurements requested by q are offered by P.

Lemma 5.10 (Relevance) *A publisher P with view $\sigma_D(r)$, where $D = D^\kappa \wedge D^\mu$, is* relevant *for a query $\sigma_C(r)$ with $C = C^\kappa \wedge C^\mu$ if it has the following two properties.*

1. *$C \wedge D$ is satisfiable (Consistency).*

2. *$C^\mu \models D^\mu$ (Measurement Entailment).*

Clearly, strict relevance implies relevance. Also, a relevant republisher may become strictly relevant if the right producers are added to the current configuration.

5.3.2 Subsumption of Publishers

In principle, there is a wide range of possibilities to construct query plans in the presence of republishers. It is desirable in the proposed stream integration system, and in the context of R-GMA, to give preference to republishers over producers, since one of the main reasons for setting up republishers is to support more efficient query answering. Among the republishers, preference is given to those that can contribute as many channels as possible to a query. In order to be able to rank publishers a subsumption relationship is introduced.

A stream s_1 is *subsumed* by a stream s_2 if for every channel c_1 in s_1 there is a channel c_2 in s_2 such that c_1 is a substream of c_2. A publisher P is *subsumed* by a republisher R with respect to a configuration $\mathcal{P}$, if for every instance $\mathcal{J}$ of $\mathcal{P}$ the stream $P^\mathcal{J}$ is subsumed by $R^\mathcal{J}$. Since $\mathcal{P}$ is usually clear from the context, this is simply denoted as $P \preceq R$. Publisher P is *strictly subsumed* by R written as $P \prec R$ if P is subsumed by R but not vice versa.

The definition entails that if P has the view $\sigma_{D^\kappa \wedge D^\mu}(r)$ and R the view $\sigma_{E^\kappa \wedge E^\mu}(r)$, then P is subsumed by R if and only if

$$D^\kappa \models E^\kappa \qquad \text{and} \qquad E^\mu \models D^\mu. \tag{5.23}$$

When considering a query $q = \sigma_C(r)$, where $C = C^\kappa \wedge C^\mu$, it is desirable to rank the relevant publishers for q according to the channels they can contribute to q. If P is a relevant publisher for q and R a relevant republisher, then P is *subsumed by R w.r.t. q*, written as $P \preceq_q R$, if for every instance $\mathcal{J}$ of $\mathcal{P}$ the stream $\sigma_C(P^\mathcal{J})$ is subsumed by $\sigma_C(R^\mathcal{J})$. The notation $P \prec_q R$ is used to express that P is strictly subsumed by R w.r.t. q.

If the descriptive view of P is $\sigma_{D^\kappa \wedge D^\mu}(r)$ and the one of R is $\sigma_{E^\kappa \wedge E^\mu}(r)$, then $P \preceq_q R$ if and only if

$$D^\kappa \wedge C^\kappa \models E^\kappa. \tag{5.24}$$

The property $C^\mu \wedge E^\mu \models C^\mu \wedge D^\mu$ is always satisfied, since the relevance of R and P implies that $C^\mu \models E^\mu$ and $C^\mu \models D^\mu$.

5.3.3 Plans Using Maximal Relevant Republishers

A method for constructing query plans that consist of publishers that are maximal relevant with regard to the subsumption relation "$\preceq_q$" is presented. During the construction of a query plan it is assumed that the publisher configuration and the query $q = \sigma_C(r)$ are fixed.

A relevant publisher is *maximal* if it is not strictly subsumed by another relevant publisher. Let M_q be the set of maximal relevant publishers for q. The set M_q is partitioned into the subsets $M_q^\mathbf{S}$ and $M_q^\mathbf{R}$, consisting of stream producers and republishers, respectively.

If $P_1 \preceq_q P_2$ and $P_2 \preceq_q P_1$ then $P_1 \sim_q P_2$. Clearly, if P_1 and P_2 are two distinct maximal relevant publishers, and $P_1 \preceq_q P_2$ then $P_1 \sim_q P_2$. Note that a producer is never equivalent to another publisher because it cannot subsume the other publisher. Thus, the relation "$\sim_q$" is an equivalence relation on the set of republishers $M_q^\mathbf{R}$ and R_1 is *equivalent to R_2 w.r.t. q* if $R_1 \sim_q R_2$.

The equivalence class of a republisher R w.r.t. q is denoted as $[R]_q$. Any two equivalent republishers will contribute the same answer tuples satisfying q. Therefore, only one element of any class $[R]_q$ needs to be chosen when constructing a plan for q. The set of all equivalence classes of maximal relevant republishers is denoted as

$$\mathcal{M}_q^\mathbf{R} = \{\, [R]_q \mid R \in M_q^\mathbf{R} \,\}. \tag{5.25}$$

The pair $\mathcal{M}_q = (\mathcal{M}_q^{\mathbf{R}}, M_q^{\mathbf{S}})$ is the *meta query plan* for q. The meta query plan can then be used to construct a valid query plan for the query q.

A sequence $\langle R_1, \ldots, R_k \rangle$ of republishers that is obtained by choosing one representative from each class of republishers in $\mathcal{M}_q^{\mathbf{R}}$ is called a *supplier sequence* for q. Let $\langle R_1, \ldots, R_k \rangle$ be a supplier sequence for q and $S_1, \ldots, S_l$ be the stream producers in $M_q^{\mathbf{S}}$. Suppose the descriptive views of the R_i have the conditions D_i. The *canonical republisher query* for the sequence is defined as

$$Q^{\mathbf{R}} = \sigma_{C_1}(R_1) \uplus \cdots \uplus \sigma_{C_k}(R_k), \tag{5.26}$$

where $C_1 = C$ and $C_i = C \ \wedge \ \neg(D_1 \vee \cdots \vee D_{i-1})$ for $i \in 2..k$. Similarly the *canonical stream producer query* is defined as

$$Q^{\mathbf{S}} = \sigma_{C'}(S_1) \uplus \cdots \uplus \sigma_{C'}(S_l), \tag{5.27}$$

where $C' = C \ \wedge \ \neg(D_1 \vee \ldots \vee D_k)$.

The selection conditions on the disjuncts in $Q^{\mathbf{R}}$ ensure that R_i only contributes channels that no $R_{i'}$ with $i' < i$ can deliver, and the condition C' in $Q^{\mathbf{S}}$ guarantees that producers only contribute channels that cannot be delivered by the republishers.

Note that the conditions C_i depend on the order of republishers in the sequence, but once the order is fixed, they do not depend on which republisher is chosen from an equivalence class. Moreover, although syntactically the conditions C' in $Q^{\mathbf{S}}$ may differ for different supplier sequences, they are all equivalent.

Theorem 5.11 *Let q be a global query, $Q^{\mathbf{R}}$ be the canonical republisher query, and $Q^{\mathbf{S}}$ be the canonical stream producer query for some supplier sequence for q. Then, a plan for q is*

$$Q = Q^{\mathbf{R}} \uplus Q^{\mathbf{S}}. \tag{5.28}$$

Proof. (Sketch) To prove that Q is a plan, it is required to show that Q is sound and complete for q, duplicate free and weakly ordered.

The conditions in the selections of Q satisfy (5.10) and thus ensure soundness. They also satisfy (5.17) and thus ensure duplicate freeness. Completeness is guaranteed because Q satisfies the properties stated in Theorem 5.3 because maximal republishers are chosen for Q, together with producers that are not subsumed by

a republisher. Finally, Q is weakly ordered because the republishers used in Q are relevant and thus satisfy the Measurement Entailment Property. □

The query planning mechanism presented will now be illustrated by considering two example queries on the network throughput (ntp) relation. As previous presented in (4.1) and (5.1), the schema for the ntp relation is

$$\mathsf{ntp}(\underline{\mathsf{from}}, \underline{\mathsf{to}}, \underline{\mathsf{tool}}, \underline{\mathsf{psize}}, \mathsf{latency}, [\mathsf{timestamp}]). \tag{5.29}$$

The publisher configuration $\mathcal{P}_0$ for the example consists of five producers and three republishers. The producers are located at three sites: Heriot-Watt (hw), Rutherford Appleton Laboratory (ral), and London (lon). There are two separate tools used for generating the network throughput measures: the pinger tool and a UDP monitoring tool. The views registered by the producers are:

$$S_1 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'udpmon'}}(\mathsf{ntp}) \qquad S_2 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp})$$

$$S_3 := \sigma_{\mathsf{from}=\texttt{'ral'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp}) \qquad S_4 := \sigma_{\mathsf{from}=\texttt{'ral'} \wedge \mathsf{tool}=\texttt{'udpmon'}}(\mathsf{ntp})$$

$$S_5 := \sigma_{\mathsf{from}=\texttt{'lon'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp})$$

The republishers that are registered are

$$R_1 := \sigma_{\mathsf{from}=\texttt{'hw'}}(\mathsf{ntp}) \qquad R_2 := \sigma_{\mathsf{from}=\texttt{'ral'}}(\mathsf{ntp})$$

$$R_3 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp}).$$

Republisher R_1 collects all the ntp measurements which originate at the Heriot-Watt site. Similarly, republisher R_2 collects all the measurements originating at Rutherford Appleton Laboratory. Republisher R_3 collects all the ntp measurements originating at Heriot-Watt that were made using the pinger tool. It is assumed that the publishers organise themselves into a suitable structure. The interconnections between the publishers are shown in Figure 5.1.

First consider the query

$$q_1 := \sigma_{\mathsf{from} = \texttt{'hw'} \ \wedge \ \mathsf{psize} \geq 1024}(\mathsf{ntp}), \tag{5.30}$$

which retrieves all measurements originating at Heriot-Watt which were generated for a packet size greater than 1024 bytes. The first stage in the query planning process is

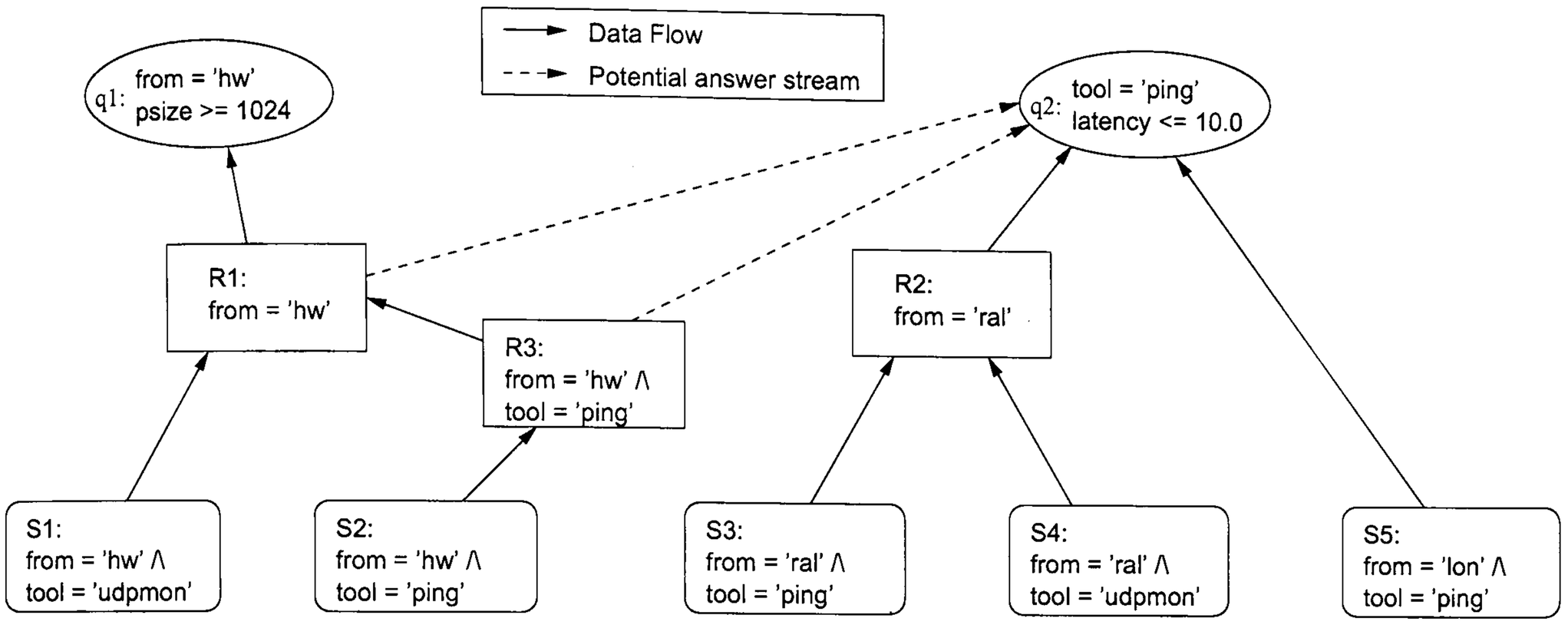


Figure 5.1: Publisher configuration $\mathcal{P}_0$ with the plans derived for queries q_1 and q_2.

to find the set of relevant publishers. This is the set

$$\{ S_1, S_2, R_1, R_3 \}. \tag{5.31}$$

The next stage is to identify which of the relevant publishers are maximal and to group them into the sets of maximal relevant republishers $M_{q_1}^{\mathbf{R}}$ and maximal relevant producers $M_{q_1}^{\mathbf{S}}$. The resulting sets are

$$M_{q_1}^{\mathbf{R}} = \{ R_1 \} \tag{5.32}$$

$$M_{q_1}^{\mathbf{S}} = \emptyset. \tag{5.33}$$

Since there is only one maximal relevant publisher for q_1 the rest of the query planning is straightforward. The meta query plan will contain just one equivalence class with a single element. This one maximal relevant publisher will then be used to form the query plan resulting the plan

$$Q_1 = \sigma_{\mathsf{from} = \texttt{'hw'} \ \wedge \ \mathsf{psize} \geq 1024}(R_1), \tag{5.34}$$

which selects all the answer tuples required for the query from the republisher R_1. The execution of this query plan is shown in Figure 5.1 by the solid line from R_1 to the consumer with the query q_1.

Now consider the consumer with the query

$$q_2 := \ \sigma_{\mathsf{tool} = \texttt{'ping'} \ \wedge \ \mathsf{latency} \leq 10.0}(\mathsf{ntp}), \tag{5.35}$$

which asks for all measurements made with the pinger tool which had a latency less than or equal to 10.0 seconds.

The set of relevant publishers for q_2 is

$$\{ S_2, S_3, S_5, R_1, R_2, R_3 \}. \tag{5.36}$$

From the set of relevant publishers, the sets of maximal relevant publishers are derived as

$$M_{q_2}^{\mathbf{R}} = \{ R_1, R_2, R_3 \} \tag{5.37}$$

$$M_{q_2}^{\mathbf{S}} = \{ S_5 \}. \tag{5.38}$$

The set $M_{q_2}^{\mathbf{R}}$ is then used to generate equivalence classes of republishers for the query. The equivalence classes, along with the set $M_{q_2}^{\mathbf{S}}$, are then used to form the meta query plan

$$\mathcal{M}_{q_2} = \Big(\{ \{ R_1, R_3 \}, \{ R_2 \} \}, \{ S_5 \} \Big). \tag{5.39}$$

Several query plans consistent with $\mathcal{M}_{q_2}$ are possible. However, they essentially require a choice in whether to consume from republisher R_1 or republisher R_3. The possible connections resulting from the query plans are shown in Figure 5.1, the dotted lines representing the choice that needs to be made.

The next stage in the query planning process is to generate a supplier sequence. For the sake of the example, the following sequence shall be used,

$$\langle R_1, R_2, S_5 \rangle. \tag{5.40}$$

This supplier sequence is then used to form the query plan

$$\begin{aligned} Q_2 = {} & \sigma_{\mathsf{tool} \,=\, \texttt{'ping'} \;\wedge\; \mathsf{latency} \,\leq\, 10.0}(R_1) \uplus \\ & \sigma_{\mathsf{tool} \,=\, \texttt{'ping'} \;\wedge\; \mathsf{latency} \,\leq\, 10.0 \,\wedge\; \neg(\mathsf{from}=\texttt{'hw'})}(R_2) \uplus \\ & \sigma_{\mathsf{tool} \,=\, \texttt{'ping'} \;\wedge\; \mathsf{latency} \,\leq\, 10.0 \,\wedge\; \neg(\mathsf{from} \,=\, \texttt{'hw'} \,\vee\, \mathsf{from} \,=\, \texttt{'ral'})}(S_5). \end{aligned} \tag{5.41}$$

5.3.4 Discussion

The computation of query plans that use maximal relevant republishers involves satisfiability and entailment checks. Clearly, this makes the task intractable in the worst case if arbitrary conditions are permitted. However, if conditions in queries and views are of the restricted form that have been considered here, namely conjunctions of the form

$$attr_1 \, op_1 \, val_1 \;\wedge\; \ldots \;\wedge\; attr_n \, op_n \, val_n,$$

where $op_i \in \{<, \leq, =, \geq, >\}$, then both satisfiability and entailment checks are polynomial. They remain polynomial if slightly more general conditions are allowed by admitting also comparisons between attributes of the form "$attr_1$ *op* $attr_2$" or limited disjunctions of the form "$attr$ in $\{ val_1, \ldots, val_n \}$".

The query planning technique presented can be used for planning consumer queries in the stream integration system. However, for planning republisher queries some modifications are needed to ensure that plans do not introduce cyclic dependencies between republishers. These techniques will be discussed in Chapter 6. Implementation decisions such as where to perform each part of the query planning tasks will be discussed in Chapter 7.

5.4 Summary

This chapter has presented a formal model for a data stream. Properties and operations have been defined that streams conforming to this model may have. The model was then used to formalise the publication and querying of streams of data that would be found in an implementation of the stream integration system proposed in Chapter 4.

A key component of the stream integration system is the republisher. However, these introduce redundancy in the data and mean that a choice must be made as to where to retrieve data for a query. This choice is made by the plan for the query. Four desirable properties, soundness and completeness with respect to a query, duplicate freeness, and weak order, for a query plan were identified and defined.

Finally, a mechanism was presented for generating query plans for consumer queries in the presence of republishers that were guaranteed to produce answer streams with the four desirable properties. The next chapter will consider how these query plans are affected by changes to the publisher configuration and how the mechanism can be altered to accommodate republisher queries.

Chapter 6

Maintaining Query Plans

A continuous query is expected to be executed over a long period of time. Over time, the publisher configuration will change as new publishers are created, or existing ones are removed or fail. Thus, the query plan for a long-lived continuous query will need to be amended to reflect the changes in the publisher configuration in order to maintain the desirable properties of a query plan. Failure to update the query plans could result in a query not receiving the complete answer stream and in the worst case would result in no answer stream being returned.

This chapter begins by considering how the query plan for a consumer query can be *maintained* when the publisher configuration changes, i.e. how the query plan can be amended to reflect a new publisher configuration without needing to replan the query from scratch.

Section 6.2 goes on to consider the problem of generating query plans for the queries posed by the republishers. These should guarantee the four desirable properties of soundness and completeness with respect to the query, duplicate freeness, and weak order, but must also ensure other properties to provide answer streams in an efficient manner. These properties will be identified and mechanisms to generate and maintain republisher query plans detailed.

The work in this chapter has been published in [112, 113].

6.1 Maintaining Consumer Query Plans

When there is a change in the publisher configuration, the query plans of consumers must be adapted to the new situation to ensure the properties for the answer stream. A meta query plan $\mathcal{M}_q$, as defined in Chapter 5, depends on two parameters: a global query q, which is explicit in the notation, and a publisher configuration $\mathcal{P}$, which so far was taken to be fixed. However, during the execution period of a global query which is long lived, it is possible that $\mathcal{P}$ changes to a new configuration $\mathcal{P}'$ because new publishers arise or existing ones vanish. As a consequence, the meta query plan for q in the new configuration $\mathcal{P}'$ may differ from the one in $\mathcal{P}$ and the query plan may have to change as well. To make the dependency on the publisher configuration explicit, in this section meta query plans for q w.r.t. $\mathcal{P}$ and $\mathcal{P}'$ shall be written as $\mathcal{M}_q(\mathcal{P})$ and $\mathcal{M}_q(\mathcal{P}')$, respectively.

One possibility to move from $\mathcal{M}_q(\mathcal{P})$ to $\mathcal{M}_q(\mathcal{P}')$ would be to compute the new meta query plan from scratch. However, it is likely that the meta query plan will remain mostly the same or not need to be changed since the difference between the two configurations is just one publisher. Therefore, it is likely to be more efficient to

1. Identify when at all $\mathcal{M}_q(\mathcal{P})$ is affected by a change of $\mathcal{P}$.

2. Amend $\mathcal{M}_q(\mathcal{P})$, whenever this is possible, based on the information contained in $\mathcal{M}_q(\mathcal{P})$ and the publisher involved in the change.

The rest of this section shall investigate formally how *adding* a publisher to $\mathcal{P}$ or *deleting* one affects meta query plans. Without loss of generality, it is assumed that all publishers added to or deleted from $\mathcal{P}$ are relevant for q, since other changes do not have an effect on the meta query plan.

As a running example through this section the publisher configuration $\mathcal{P}_0$, which was introduced in Section 5.3.3, for the network throughput (ntp) relation will be considered. Again, the schema consists of

$$\mathsf{ntp}(\underline{\mathsf{from}}, \underline{\mathsf{to}}, \underline{\mathsf{tool}}, \underline{\mathsf{psize}}, \mathsf{latency}, [\mathsf{timestamp}]). \tag{6.1}$$

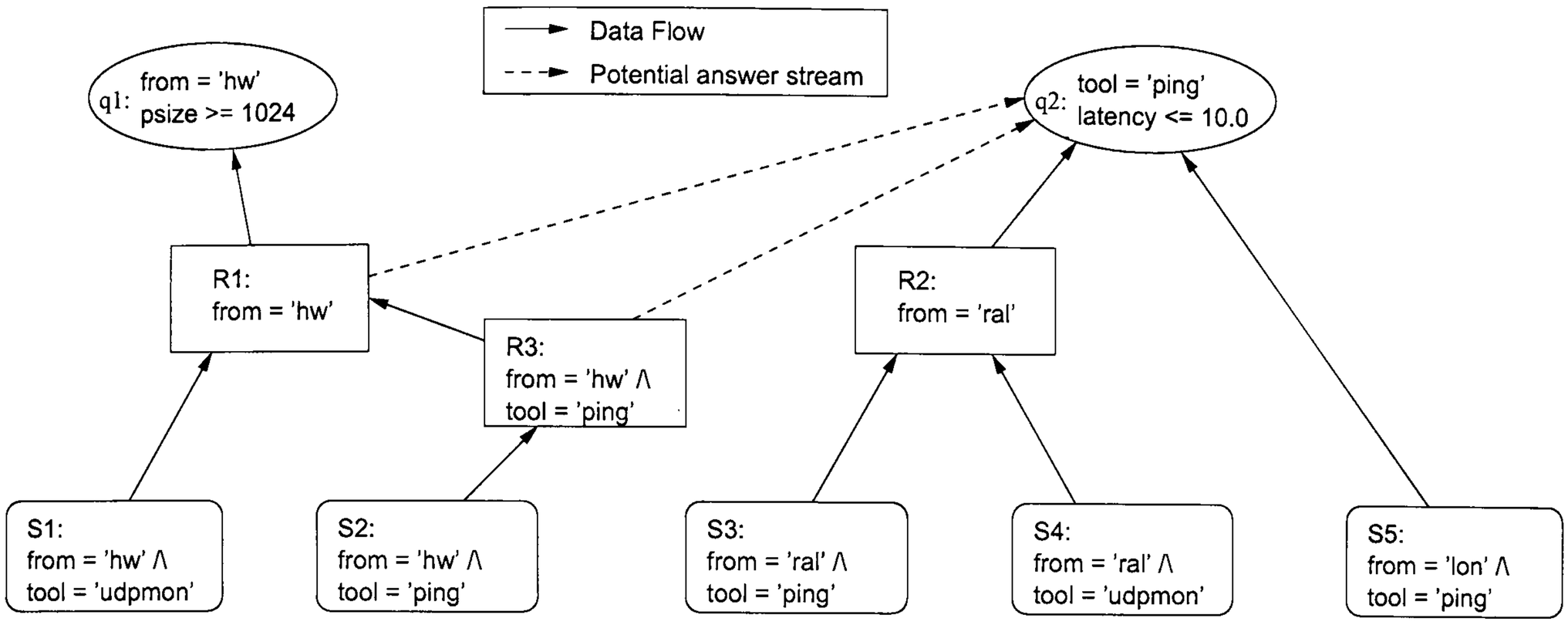


Figure 6.1: Publisher configuration $\mathcal{P}_0$ with the plans derived for queries q_1 and q_2.

The publisher configuration consists of the publishers

$$S_1 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'udpmon'}}(\mathsf{ntp}) \qquad S_2 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp})$$

$$S_3 := \sigma_{\mathsf{from}=\texttt{'ral'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp}) \qquad S_4 := \sigma_{\mathsf{from}=\texttt{'ral'} \wedge \mathsf{tool}=\texttt{'udpmon'}}(\mathsf{ntp})$$

$$S_5 := \sigma_{\mathsf{from}=\texttt{'lon'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp})$$

$$R_1 := \sigma_{\mathsf{from}=\texttt{'hw'}}(\mathsf{ntp}) \qquad R_2 := \sigma_{\mathsf{from}=\texttt{'ral'}}(\mathsf{ntp})$$

$$R_3 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp}).$$

Again, the consumer queries

$$q_1 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{psize} \geq 1024}(\mathsf{ntp}), \text{ and}$$

$$q_2 := \sigma_{\mathsf{tool}=\texttt{'ping'} \wedge \mathsf{latency} \leq 10.0}(\mathsf{ntp}),$$

are considered. Figure 6.1 illustrates the data connections that were assumed for the republishers along with the query plans derived by the planning mechanism detailed in Chapter 5.

6.1.1 Adding a Producer

If a relevant producer S_0 is added then there are two cases to be considered. If S_0 is subsumed w.r.t. q by an existing maximal republisher, say R, then all the data coming from S_0 will be republished by R and, similarly, by every republisher in $[R]_q$, the equivalence class of R. Since the current meta query plan contains the class $[R]_q$,

no change is needed. However, if S_0 is not subsumed by a maximal republisher, then it has to be added to the set of maximal relevant producers.

Proposition 6.1 *Suppose* $\mathcal{P}' = \mathcal{P} \cup \{ S_0 \}$. *Then:*

1. *If there is a class* $[R]_q \in \mathcal{M}_q^{\mathbf{R}}$ *such that* $S_0 \preceq_q R$, *then* $\mathcal{M}_q(\mathcal{P}') = \mathcal{M}_q(\mathcal{P})$;
2. *If there is no such class, then* $\mathcal{M}_q(\mathcal{P}') = (\mathcal{M}_q^{\mathbf{R}}, M_q^{\mathbf{S}} \cup \{ S_0 \})$.

Proof. The only change in the publisher configuration from $\mathcal{P}$ to $\mathcal{P}'$ is the addition of the producer S_0. Since the republisher configuration has not changed then $\mathcal{M}_q^{\mathbf{R}}$ will remain unchanged.

If S_0 is not maximal relevant for q then there must exist a republisher R that is maximal relevant for q in $\mathcal{P}'$ such that $S_0 \preceq_q R$. Since $\mathcal{M}_q^{\mathbf{R}}$ is unchanged from $\mathcal{P}$ then R is maximal relevant for q in $\mathcal{P}$ and appears in $\mathcal{M}_q(\mathcal{P})$. Thus, $\mathcal{M}_q(\mathcal{P}) = \mathcal{M}_q(\mathcal{P}')$.

On the other hand, if S_0 is maximal relevant for q in $\mathcal{P}'$ then there does not exist a republisher R in $\mathcal{P}'$ such that $S_0 \preceq_q R$ and as such S_0 should appear in $\mathcal{M}_q^{\mathbf{S}}$. □

For the running example, consider adding a producer that publishes details about PingER messages originating at Glasgow to the configuration $\mathcal{P}_0$. The view registered by the new producer is

$$S_6 := \sigma_{\mathsf{from='gla'} \wedge \mathsf{tool='ping'}}(\mathsf{ntp}). \tag{6.2}$$

Since S_6 is not relevant for q_1, due to the unsatisfiability of their conditions, there is no effect on either the meta query plan or the query plan of q_1. Producer S_6 is relevant for q_2. The situation is that of case 2 of Proposition 6.1, since there are no republishers that would collect the data published by S_6. In this instance the producer would be added to both the meta query plan and query plan of q_2. The resulting configuration $\mathcal{P}_1$ is shown in Figure 6.2.

6.1.2 Deleting a Producer

If a relevant producer S_0 is dropped, then the situation is similar to the previous one. If S_0 is not a maximal relevant producer, i.e. if it is subsumed by some republisher, then the meta query plan is not affected by the change. Otherwise it has to be removed from the set of maximal relevant producers.

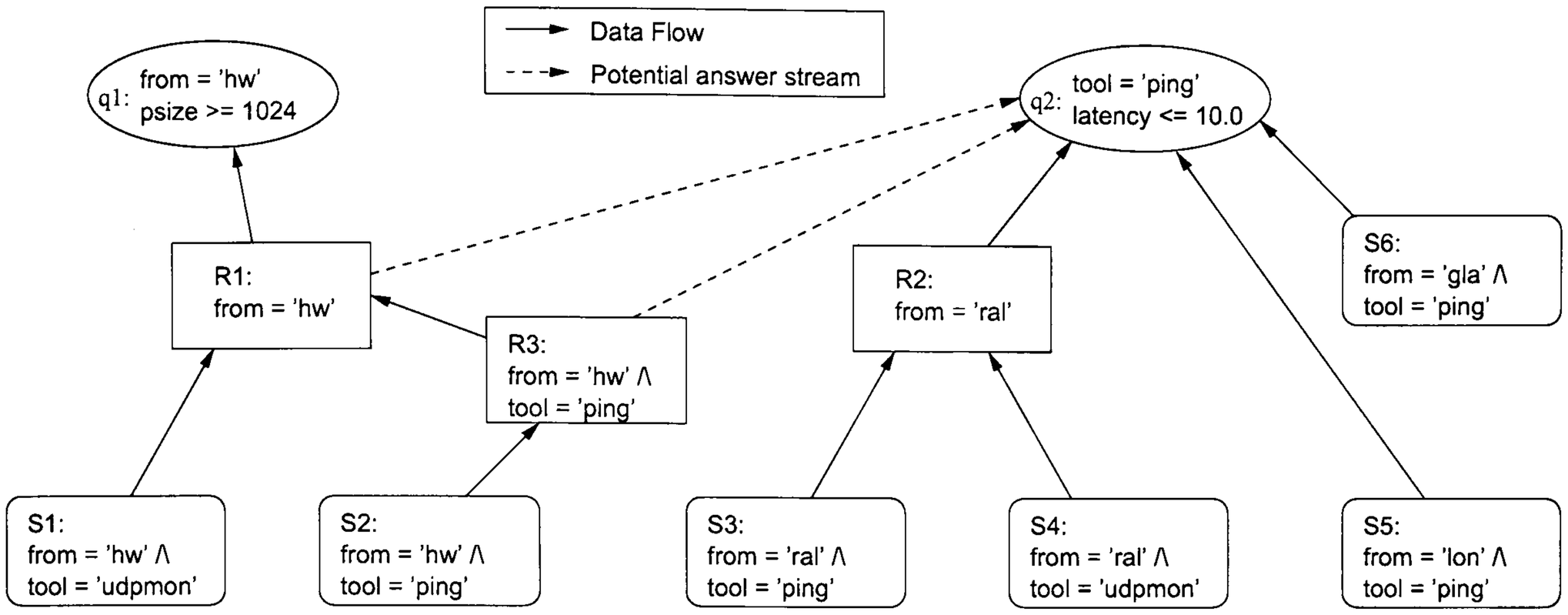


Figure 6.2: The data connections of Publisher Configuration $\mathcal{P}_1$ and the effects on queries q_1 and q_2.

Proposition 6.2 *Suppose* $\mathcal{P}' = \mathcal{P} \setminus \{ S_0 \}$. *Then:*

1. *If* $S_0 \notin M_q^{\mathbf{S}}$, *then* $\mathcal{M}_q(\mathcal{P}') = \mathcal{M}_q(\mathcal{P})$;

2. *If* $S_0 \in M_q^{\mathbf{S}}$, *then* $\mathcal{M}_q(\mathcal{P}') = (\mathcal{M}_q^{\mathbf{R}}, M_q^{\mathbf{S}} \setminus \{ S_0 \})$.

Proof. The only change in the publisher configuration from $\mathcal{P}$ to $\mathcal{P}'$ is the removal of the producer S_0. Since the republisher configuration has not changed then $\mathcal{M}_q^{\mathbf{R}}$ will remain unchanged.

If S_0 is not maximal relevant for q in $\mathcal{P}$ then S_0 does not appear in $\mathcal{M}_q(\mathcal{P})$. Since the removal of S_0 is the only change in the publisher configuration then $\mathcal{M}_q(\mathcal{P}) = \mathcal{M}_q(\mathcal{P}')$.

On the other hand, if S_0 is maximal relevant for q in $\mathcal{P}$ then it appears in $\mathcal{M}_q^{\mathbf{S}}(\mathcal{P})$. Since producers do not have any effect on the maximal relevance of any other publisher the only change required to the query plan is to remove S_0 from $\mathcal{M}_q^{\mathbf{S}}(\mathcal{P})$. □

This will be illustrated with the running example by removing producer S_6 from $\mathcal{P}_1$ to return to configuration $\mathcal{P}_0$. Since S_6 is not relevant for q_1 there is no effect. For q_2 producer S_6 was maximal relevant so case 2 of Proposition 6.2 applies. Thus, S_6 is removed from the meta query plan and query plan of q_2.

6.1.3 Adding a Republisher

The situation becomes more complex when a relevant republisher R_0 is added. There are three possible cases to be considered:

1. R_0 is strictly subsumed by some maximal republisher R, i.e. $R_0 \prec_q R$;

2. R_0 is equivalent to some maximal republisher R, i.e. $R_0 \sim_q R$; or

3. R_0 is not subsumed by any existing maximal republisher.

In case 1, R_0 is not needed in the meta query plan, while in case 2, R_0 needs to be added to the class $[R]_q$. In case 3, R_0 will form a new equivalence class of its own. Moreover, it may be the case that R_0 subsumes some existing maximal relevant producers and republishers. If it does, then the subsumption is strict and the publishers concerned have to be removed from the meta query plan.

Proposition 6.3 *Suppose* $\mathcal{P}' = \mathcal{P} \cup \{ R_0 \}$. *Then:*

1. *If there is a class* $[R]_q \in \mathcal{M}_q^{\mathbf{R}}$ *such that* $R_0 \prec_q R$, *then* $\mathcal{M}_q(\mathcal{P}') = \mathcal{M}_q(\mathcal{P})$;

2. *If there is a class* $[R]_q \in \mathcal{M}_q^{\mathbf{R}}$ *such that* $R_0 \sim_q R$, *then* $\mathcal{M}_q(\mathcal{P}')$ *is obtained from* $\mathcal{M}_q = (\mathcal{M}_q^{\mathbf{R}}, M_q^{\mathbf{S}})$ *by replacing the class* $[R]_q$ *in* $\mathcal{M}_q^{\mathbf{R}}$ *with* $[R]_q \cup \{R_0\}$;

3. *If there is no class* $[R]_q \in \mathcal{M}_q^{\mathbf{R}}$ *with* $R_0 \preceq_q R$, *then* $\mathcal{M}_q(\mathcal{P}') = (\mathcal{M}_q^{\mathbf{R}'}, M_q^{\mathbf{S}'})$ *where*

   - $\mathcal{M}_q^{\mathbf{R}'}$ *is obtained from* $\mathcal{M}_q^{\mathbf{R}}$ *by* adding *the class* $\{ R_0 \}$ *and* removing *all classes* $[R']_q$ *with* $R' \preceq_q R_0$
   - $M_q^{\mathbf{S}'}$ *is obtained from* $M_q^{\mathbf{S}}$ *by* only keeping *the producers that are not subsumed by* R_0, *i.e.*

$$M_q^{\mathbf{S}'} = \{ S \in M_q^{\mathbf{S}} \mid S \not\preceq_q R_0 \}.$$

Proof. The argument for case 1 is similar to that of adding a producer that is not maximal relevant for q.

In case 2 there exists a $[R]_q \in \mathcal{M}_q(\mathcal{P})$ such that $R_0 \sim_q R$. From the definition of equivalence any publisher subsumed by R_0 will be subsumed by R. Thus, the only change to the meta query plan is to add R_0 to the equivalence containing R.

In case 3 there does not exist an equivalence class $[R]_q \in \mathcal{M}_q(\mathcal{P})$ such that $R_0 \preceq_q R$. This implies that R_0 is maximal relevant for q in $\mathcal{P}'$, not equivalent to any other publisher and should be in an equivalence class where it is the only member.

Next, it is shown that a publisher P where $P \in \mathcal{M}_q(\mathcal{P})$ may no longer be maximal. By the assumption of this case, R_0 is not subsumed by any publisher $P \in \mathcal{P}$ but this does not preclude $P \preceq_q R_0$. This subsumption is strict since R_0 is not subsumed w.r.t. q. Thus, P is not maximal relevant for q in $\mathcal{P}'$ and not a member of $\mathcal{M}_q(\mathcal{P}')$.

Finally it is shown that a publisher P which was not maximal in $\mathcal{P}$ remains so in $\mathcal{P}'$. Since P is not maximal relevant then there exists a republisher $R' \in \mathcal{M}_q(\mathcal{P})$ such that $P \prec_q R'$. If $R' \in \mathcal{M}_q(\mathcal{P}')$ then it is still the case that $P \prec_q R'$. Otherwise, for R' not to be in $\mathcal{M}_q(\mathcal{P}')$ it must be the case that $R' \prec_q R_0$. Since subsumption is a transitive relation it follows that $P \prec_q R_0$. Thus, no publisher other than R_0 can be added to $\mathcal{M}_q(\mathcal{P})$ to form $\mathcal{M}_q(\mathcal{P}')$. □

Consider again the publisher configuration $\mathcal{P}_0$ and the effect of adding the republisher

$$R_4 := \sigma_{true}(\mathsf{ntp}), \tag{6.3}$$

which gathers all tuples published for the ntp relation. With regard to the consumer query q_1, case 2 holds since $R_4 \sim_{q_1} R_1$. Thus, R_4 would be added to the equivalence class of R_1 in $\mathcal{M}_{q_1}$. There is no need to change the query plan of q_1 as the old plan is consistent with the new meta query plan. For q_2, case 3 holds as R_4 subsumes with respect to q_2 all the maximal relevant publishers. The new meta query plan would be derived by dropping all the current equivalence classes and maximal producers and adding the new equivalence class $[R_4]_{q_2}$. Obviously the query plan of q_2 will also need to be updated to make it consistent with the new meta query plan. Techniques to switch between query plans are discussed in Section 7.4. The situation in the resulting configuration $\mathcal{P}_2$ is illustrated in Figure 6.3.

6.1.4 Deleting a Republisher

Similar to the previous situation, three cases can be distinguished when a republisher R_0 is dropped:

1. R_0 is strictly subsumed by some maximal republisher R, i.e. $R_0 \prec_q R$;

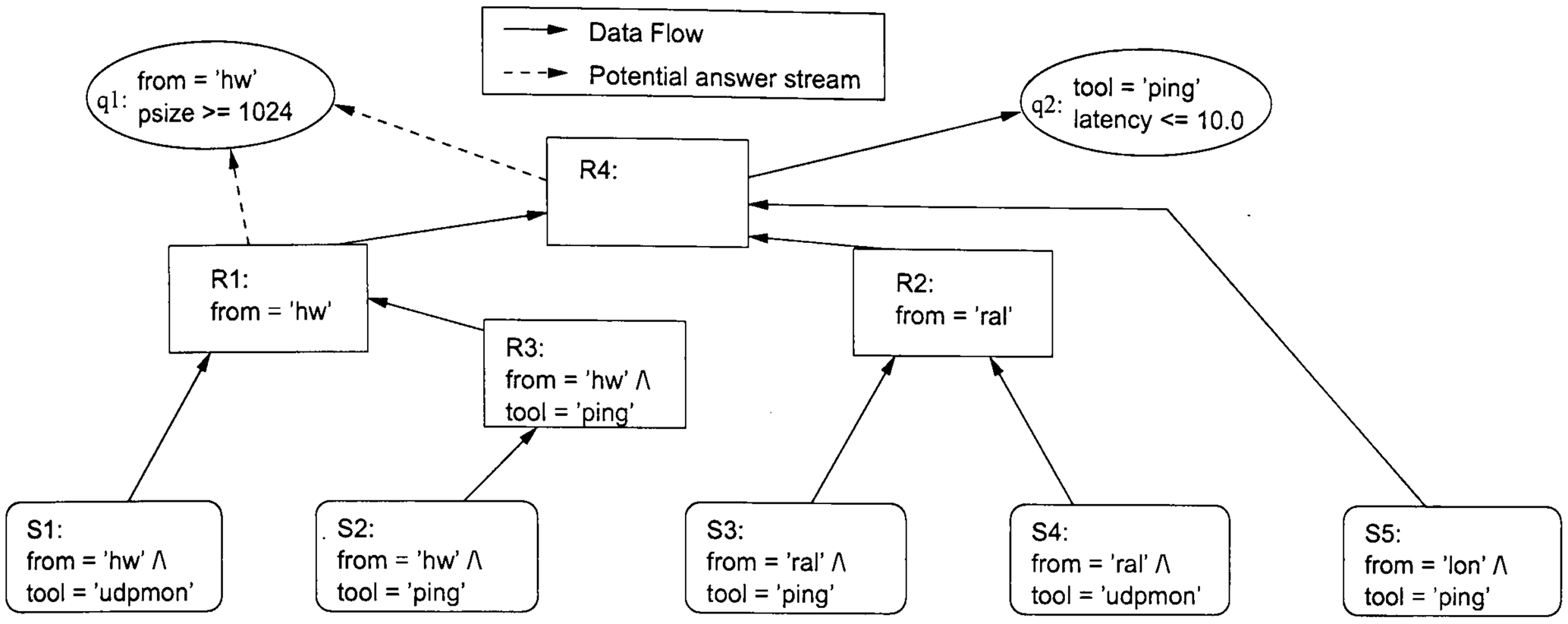


Figure 6.3: Consumer queries q_1 and q_2 being posed at publisher configuration $\mathcal{P}_2$.

2. R_0 is equivalent to some other maximal republisher R, i.e. $R_0 \sim_q R$; or

3. R_0 is not subsumed by any existing maximal republisher.

In case 1 the meta query plan is not affected, while in case 2 the republisher R_0 needs to be deleted from its equivalence class. Case 3, by contrast, requires more action. The reason is that, intuitively, the deletion of R_0 leaves a hole in the set of data that can be delivered by the remaining publishers in the meta query plan.

To "patch" the hole, those relevant publishers need to be identified that were not maximal relevant in the presence of R_0, but are promoted to maximal relevant ones after the removal of R_0. This is done in two stages. First the republishers are considered and then the producers. The *patch* of $M_q^{\mathbf{R}}$ for R_0 is defined as the set M' consisting of those republishers R' relevant for q where

1. It is the case that $R' \prec_q R_0$ and

2. There is no $R \in M_q^{\mathbf{R}} \setminus \{ R_0 \}$ such that $R' \prec_q R$.

Then the new set $\mathcal{M}_q^{\mathbf{R}'}$ is obtained by removing the class $[R_0]_q$ from $\mathcal{M}_q^{\mathbf{R}}$ and adding the classes obtained from the elements of M'. Secondly, some producers that were subsumed by R_0 may not be subsumed by the newly promoted maximal republishers and have to be added to the set $M_q^{\mathbf{S}}$ to yield $M_q^{\mathbf{S}'}$.

Proposition 6.4 *Suppose* $\mathcal{P}' = \mathcal{P} \setminus \{ R_0 \}$. *Then:*

1. *If* $R_0 \notin M_q^{\mathbf{R}}$, *then* $\mathcal{M}_q(\mathcal{P}') = \mathcal{M}_q(\mathcal{P})$:

2. *If* $R_0 \in M_q^{\mathbf{R}}$ *and there is another* $R \in M_q^{\mathbf{R}}$ *with* $R_0 \sim_q R$*, then* $\mathcal{M}_q(\mathcal{P}')$ *is obtained from* $\mathcal{M}_q = (\mathcal{M}_q^{\mathbf{R}}, M_q^{\mathbf{S}})$ *by replacing the class* $[R]_q$ *in* $\mathcal{M}_q^{\mathbf{R}}$ *with* $[R]_q \setminus \{R_0\}$*;*

3. *If* $R_0 \in M_q^{\mathbf{R}}$ *and the class* $[R_0]_q \in \mathcal{M}_q^{\mathbf{R}}$ *is a singleton, then* $\mathcal{M}_q(\mathcal{P}') = (\mathcal{M}_q^{\mathbf{R}'}, M_q^{\mathbf{S}'})$ *where*

   - $\mathcal{M}_q^{\mathbf{R}'}$ *is obtained from* $\mathcal{M}_q^{\mathbf{R}}$ *by* removing *the class* $\{R_0\}$ *and* adding *all classes* $[R']_q$ *such that* $R' \in M'$ *is in the patch* M' *of* $M_q^{\mathbf{R}}$ *for* R_0
   - $M_q^{\mathbf{S}'}$ *is obtained from* $M_q^{\mathbf{S}}$ *by* adding *those producers relevant for* q *that were subsumed by* R_0*, but are not subsumed by any republisher in* $\mathcal{M}_q^{\mathbf{R}'}$*.*

Proof. In case 1, $R_0 \notin M_q^{\mathbf{R}}$ implies that R_0 is not maximal relevant for q. Thus, there exists a republisher $R \in M_q^{\mathbf{R}}$ such that $R_0 \prec_q R$ and R_0 is not in $\mathcal{M}_q(\mathcal{P})$. Since the only difference between $\mathcal{P}$ and $\mathcal{P}'$ is the removal of R_0 then $\mathcal{M}_q(\mathcal{P}) = \mathcal{M}_q(\mathcal{P}')$.

In case 2, $R_0 \in M_q^{\mathbf{R}}$ and there exists another republisher $R \in M_q^{\mathbf{R}}$ with $R_0 \sim_q R$. Thus, for any publisher P where $P \preceq_q R_0$ holds then by the definition of equivalence $P \preceq_q R$ must also hold and thus both republishers will be members of the same equivalence class. Since the only difference between $\mathcal{P}$ and $\mathcal{P}'$ is the removal of R_0 it must be the case that R will remain maximal relevant for q in $\mathcal{P}'$. Hence, the equivalence class $[R]_q$ in $\mathcal{P}'$ will contain all of the republishers that were in $[R]_q$ in $\mathcal{P}$ with the exception of R_0 as it no longer exists.

In case 3 it is required to show that the result of using the patch is the same as if the query were planned from scratch. The patch M' considers republishers R' which are relevant for q such that

1. It is the case that $R' \prec_q R_0$, and

2. There does not exist a republisher $R \in M_q^{\mathbf{R}} \setminus \{R_0\}$ such that $R' \prec_q R$.

Suppose there exists a republisher R_1 such that $R_1 \notin M_q^{\mathbf{R}}(\mathcal{P})$, $R_1 \in M_q^{\mathbf{R}}(\mathcal{P}')$, and $R_1 \notin M'$.

For $R_1 \notin M_q^{\mathbf{R}}(\mathcal{P})$ it must be the case that there exists a republisher R_2 such that $R_1 \prec_q R_2$. However, for $R_1 \in M_q^{\mathbf{R}}(\mathcal{P}')$ it must be the case that R_2 no longer exists. The only republisher that no longer exists is R_0 so R_2 must be R_0.

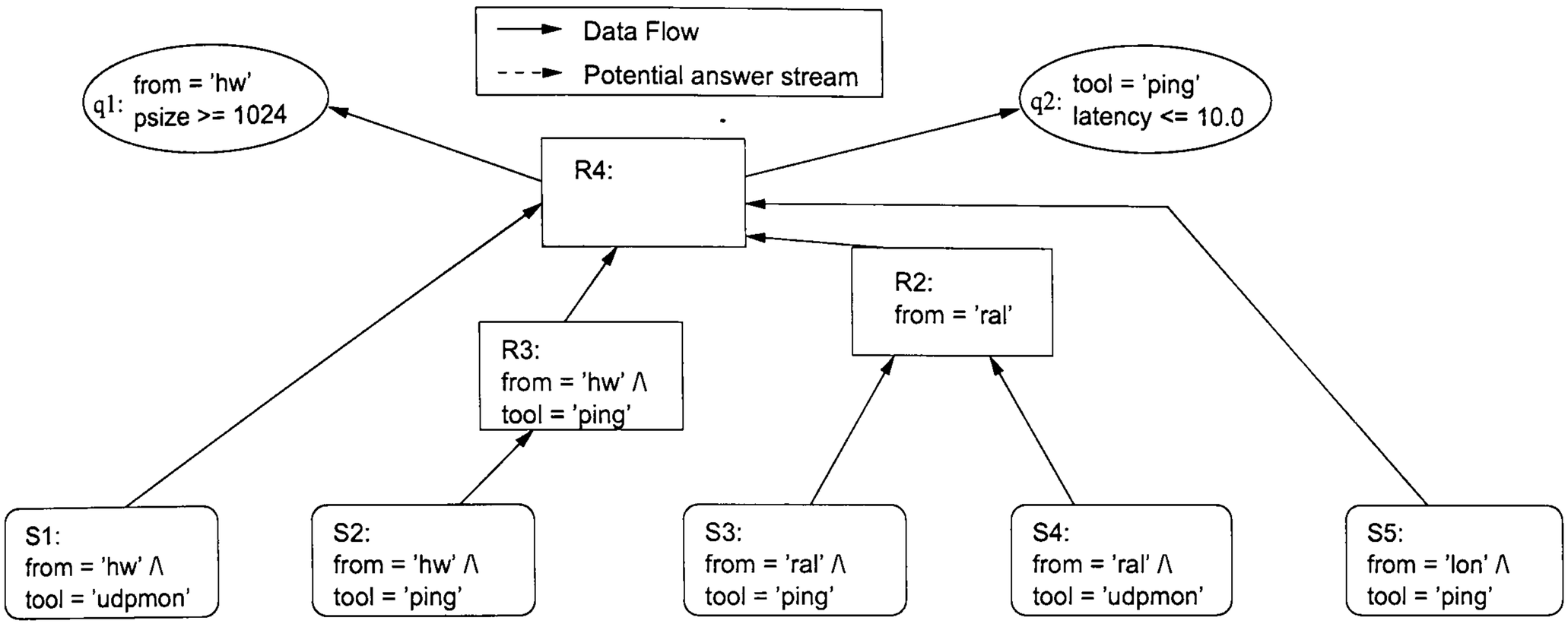


Figure 6.4: Consumer queries q_1 and q_2 being posed at publisher configuration $\mathcal{P}_3$.

For $R_1 \notin M'$ to hold it must be the case that either $R' \prec_q R_0$, or there does not exist a republisher $R \in M_q^{\mathbf{R}} \setminus \{ R_0 \}$ such that $R' \prec_q R$, do not hold. It has already been shown that the only republisher that subsumes R_1 is R_0 so this leads to a contradiction.

Therefore, it is the case that the patch contains all republishers that become maximal relevant in $\mathcal{P}'$.

All maximal relevant producers are found since all relevant producers that were not previously maximal relevant are considered. □

The situation of dropping a republisher will now be shown with the running example. Consider the case of dropping republisher R_1 from $\mathcal{P}_2$ to create $\mathcal{P}_3$. Republisher R_1 is a maximal relevant publisher for q_1 which is equivalent w.r.t. q_1 to R_4. Hence case 2 holds and the equivalence class $\{ R_1, R_4 \}$ in $\mathcal{M}_{q_1}(\mathcal{P}_2)$ is replaced with the equivalence class $\{ R_4 \}$ to give the meta query plan $\mathcal{M}_{q_1}(\mathcal{P}_3) = \left(\left\{ \{ R_4 \} \right\}, \emptyset \right)$. For q_2, republisher R_1 is not maximal relevant so case 1 of Proposition 6.4 holds and the meta query plan and query plan are unchanged.

The situation in configuration $\mathcal{P}_3$ is shown in Figure 6.4. The consumer query q_1 no longer has a choice in the publisher to contact to retrieve its answer stream. It must now contact R_4, hence the data line from R_4 to q_1 in Figure 6.4 is now solid. Note that if the consumer had been using publisher R_1 in its query plan then it would need to "switch" to a new query plan using R_4. Mechanisms to perform this switch are discussed in Section 7.4.

6.1.5 Discussion

The propositions above show that plan maintenance is straightforward if *producers* come or go and is more complicated when the set of relevant *republishers* changes. The reason is that the streams of producers are only sound with respect to their descriptive views, but republisher streams are both sound and complete. As a consequence, a republisher can replace other publishers, which is impossible for a producer. The implications of maintaining query plans during the execution of a query will be discussed in Section 7.4.

The cost of performing the plan maintenance operations is polynomial in the number of publishers involved, providing that subsumption can be checked in polynomial time. From the cases considered, and those so far encountered with the R-GMA system, this is often the case since conditions can only contain conjunctions. Of course, if conditions can contain disjunctions then the problem is NP-hard.

6.2 Planning and Maintaining Republisher Queries

In order to answer their queries efficiently, and to provide some protection from changes to the publisher configuration, consumer query plans make use of the partial answers provided by republishers. Similarly, republishers should also include other republishers in their query plans. This leads to a *hierarchy* of republishers through which data streams can flow. A straightforward approach would be to construct and maintain plans and meta plans for republishers in the same way as for consumers. However, a simple example shows that this does not work.

Consider the publisher configuration $\mathcal{P}'$ consisting of the publishers S_1, R_1, and R_4, as defined in Section 6.1. Applying the planning and maintenance techniques developed for consumer queries would result in the republishers having the meta query plans $\mathcal{M}_{R_1} = (\{\{R_4\}\}, \emptyset)$, and $\mathcal{M}_{R_4} = (\{\{R_1\}\}, \emptyset)$. The only corresponding query plans are $Q_{R_1} = \sigma_{\mathsf{from}=\text{'hw'}}(R_4)$, and $Q_{R_4} = \sigma_{true}(R_1)$.

The resulting hierarchy illustrated in Figure 6.5 is unsatisfying for two reasons.

1. The republishers are *not connected* to the producer.

2. There is a *cycle in the dependency relation* of the republishers in that R_1 con-

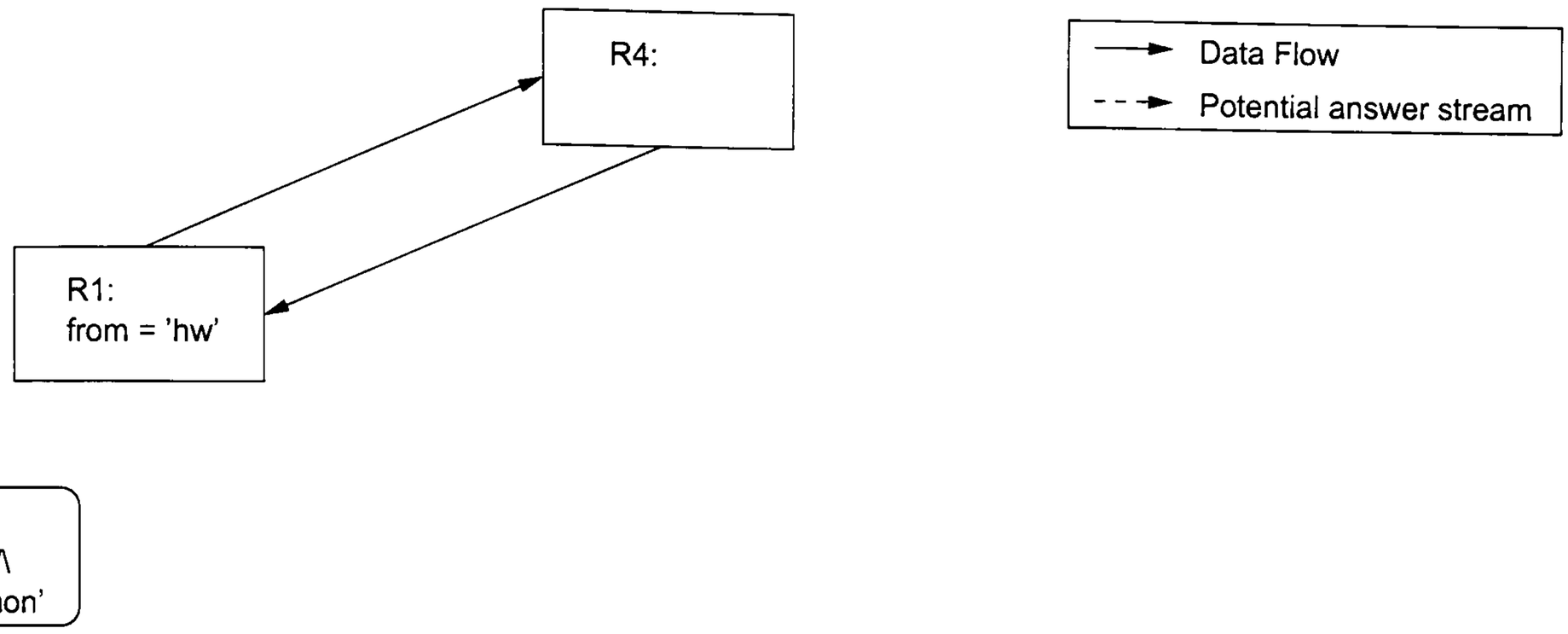


Figure 6.5: The result of using consumer planning techniques for republisher queries.

sumes from R_4 and vice versa.

Obviously, the first fact prevents the republishers from obtaining any data. Moreover, if there are cyclic dependencies between republishers, tuples could travel an infinite number of times along the cycle. The volume of the resulting stream could grow indefinitely, and the stream would be neither duplicate free nor weakly ordered.

6.2.1 Requirements of a Publisher Hierarchy

Before developing the mechanism for constructing and maintaining query plans for republisher queries, the properties that a publisher hierarchy should have are identified.

It is anticipated that query plans for republishers should be unions of selections over publishers and each republisher has a meta query plan from which the actual plans can be formed. Moreover, the mechanism should produce some kind of meta query plan that contains a set of candidate publishers on which actual plans are based.

The query plans executed by the republishers define a dependency relation among the publishers called the *physical hierarchy*, that is, one through which the data flows. The meta query plans of the republishers define a more general dependency relation called the *logical hierarchy*.

The requirements that are essential for any planning and maintenance mechanism for republishers are:

Correctness: The plan for each republisher should be sound and complete for the defining query as well as duplicate free and weakly ordered.

Cycle Freeness: Neither the physical nor the logical hierarchy should contain any cycles.

Uniqueness of the Logical Hierarchy: The logical hierarchy should only depend on the publisher configuration $\mathcal{P}$. The way in which it has been created, i.e. the order in which publishers have been added and deleted, should have no influence on it.

Local Query Planning: To create its query plan and meta query plan, a republisher should not need any information about the plans and meta plans of other republishers.

Clearly, a plan that is not correct would fail to implement the republisher and can lead to the republishers being disconnected from the producers, as in the example above.

The physical hierarchy of the example contains a cycle. The effects of the cycle are that data would continuously flow around the cycle increasing the amount of data flowing through the system. Since cycles in the logical hierarchy may give rise to cycles in the physical hierarchy, they need to be ruled out too.

A logical hierarchy will be much easier to understand if it depends only on the structure of a configuration and not on its history. Then, for a given publisher configuration, regardless of the order that the publishers were added the same logical hierarchy will be created.

For the physical hierarchy, which is a subrelation of the logical hierarchy, a republisher should be allowed to form it as it sees fit. If query planning is local, republishers only need to communicate with the registry service and not with other republishers. The physical hierarchy may depend on the order in which the publishers have been added to the system.

6.2.2 Generating and Maintaining Republisher Query Plans

A general analysis shows that cycles and missing links to producers as in the example above are a consequence of the definition of relevant publishers in Proposition 5.9. To avoid cycles, for two republishers R_1 and R_2 it should be impossible that both R_1 is relevant for R_2 and at the same time R_2 is relevant for R_1.

To refine the concept of relevance the general subsumption relation introduced in Section 5.3.2 is considered. Let P be a publisher with the view $\sigma_{D^\kappa \wedge D^\mu}(r)$ and R be a republisher with the view $\sigma_{E^\kappa \wedge E^\mu}(r)$. Publisher P is *subsumed* by R, written $P \preceq R$, if

$$D^\kappa \models E^\kappa \qquad \text{and} \qquad E^\mu \models D^\mu. \tag{6.4}$$

Intuitively, this means that R delivers tuples for a channel if P does so, and that for its channels, P delivers all the values that R delivers. Publisher P is *strictly subsumed* by R, written $P \prec R$, if $P \preceq R$, but not $R \preceq P$. Clearly, if both R_1 is subsumed by R_2 and R_2 by R_1, then the view conditions of R_1 and R_2 are logically equivalent.

This notion of general subsumption is used to modify the definition of relevance from Lemma 5.10 to that of Proposition 6.5. One readily verifies that strong relevance implies relevance.

Proposition 6.5 (Strong relevance) *A producer is* strongly relevant *if it is relevant for the query of R_0 and a republisher R is* strongly relevant *for R_0 if $R \prec R_0$.*

Reconsider the example from the beginning of Section 6.2. In $\mathcal{P}'$, both S_1 and R_1 are strongly relevant for R_4 while only S_1 is strongly relevant for R_1. If instead of relevant publishers, only strongly relevant publishers are admitted to query planning, then the meta query plan for R_1 in $\mathcal{P}'$ is $\mathcal{M}_{R_1}(\mathcal{P}') = \big(\emptyset, \{\, S_1 \,\}\big)$, while the meta query plan for R_3 is $\mathcal{M}_{R_4}(\mathcal{P}') = \big(\{\, \{\, R_1 \,\} \,\}, \emptyset\big)$. The corresponding query plans are $Q_{R_1} = \sigma_{\mathsf{from}=\text{'hw'}}(S_1)$ and $Q_{R_4} = \sigma_{true}(R_1)$. Neither the physical nor the logical hierarchy contain a cycle. The next proposition shows that this is not accidental.

Proposition 6.6 *If meta query plans for republishers are only based on strongly relevant publishers, then all plans derived from them are correct. Moreover, for any publisher configuration, there is a unique logical hierarchy, which is cycle free, and meta query plans and query plans can be computed locally by each republisher.*

Intuitively, the result holds for the following reasons. Plans are still correct, as they were for consumer queries, because the relevance criterion for producers has not been changed. A plan using only strongly relevant republishers may have to access more producers than one relying on relevant republishers, because fewer producers are made redundant by the republishers considered. By definition, the logical hierarchy

depends only on the publishers and their conditions, which uniquely determine it. Since any strongly relevant publisher for republisher R is strictly subsumed by R, there cannot be any cycles in the logical hierarchy. A plan can be computed by an individual republisher based on information about its strongly relevant publishers, without co-ordinating the planning with other republishers.

The meta query plan for a consumer or republisher query q is defined in terms of relevant or strongly relevant publishers respectively, that are maximal relevant with respect to the quasiorder "$\preceq_q$". A closer inspection of the results in Section 6.1 reveals that all four propositions hold, independently of how relevant publishers are defined. Therefore, the maintenance techniques of that section can be applied directly for republisher queries.

6.3 Summary

This chapter has considered the effect of a change in the publisher configuration on the query plan for a consumer query. It was shown that in the majority of cases, the query plan could be updated to reflect the new publisher configuration by using the information contained in the meta query plan and details of the publisher that has caused the change in the configuration. This was not possible when a republisher was being removed from the system which was the only publisher in an equivalence class in the meta query plan. While plan maintenance is only required, from the point of view of ensuring the four desirable properties of an answer stream, when a republisher is removed from the system or a producer is added, it is recommended that these techniques be followed whenever there is a change in the publisher configuration to ensure that queries are answered by using the republishers as much as possible.

The chapter then considered the problem of generating and maintaining query plans for republisher queries. It was shown that the query planning techniques developed for consumer queries were not suitable. Properties for a hierarchy of publishers were identified and it was then shown that making a small amendment to the relevance criteria would ensure that these properties held. Since the maintenance techniques for consumer query plans did not rely on the relevance criteria used, these results follow immediately for republisher queries.

The techniques developed in the last two chapters allow a stream integration system to generate and maintain query plans for selection queries over a global schema. The next chapter shall consider how to implement these techniques.

Chapter 7

Implementation Details

This chapter considers the design decisions that need to be taken in order to implement the stream integration system proposed in Chapter 4 together with the query planning and maintenance techniques developed in Chapters 5 and 6. Where necessary, these decisions are guided by the motivating application of a Grid information and monitoring system.

Section 7.1 will consider the functionality required from each of the components in the proposed stream integration system to implement the query planning and maintenance mechanisms. Much of the infrastructure required is already available in the implementation of the R-GMA system, which is a partial implementation of the proposed stream integration system. Thus, details of the R-GMA implementation will be given in Section 7.2.

In the R-GMA system only producers are considered when planning the execution of a continuous query. Section 7.3 discusses the issues in implementing the extended query planning and maintenance mechanisms to allow republishers to be used to answer continuous queries.

The final section of this chapter will consider how, in the extended implementation, the transition from one query plan to another can be managed. However, the implementation and testing of these protocols was deemed to be beyond the scope of this thesis.

7.1 Implementing the Stream Integration System

The stream integration system proposed in Chapter 4 consisted of several components, the interaction of which was shown in Figure 4.1. However, the functionality of the components with regard to query planning and maintenance was not considered. The choices that need to be made about which part of the query planning and maintenance mechanisms are performed by which components shall now be considered.

When a new continuous query is added to the system by either a consumer or a republisher, a query plan must be generated to answer the query. The query planning mechanism requires details of the publishers that are stored by the registry service together with the ability to perform the subsumption tests detailed in Section 5.3. Three ways in which the query planning can be performed by the agent responsible for the query and the registry service are:

1. The registry service generates a query plan and passes it to the agent.

2. The registry service informs the agent of all publishers registered in the system and the agent generates the query plan.

3. The registry service identifies the relevant publishers for the continuous query, passes their details to the agent, and the agent generates the query plan.

In the first case, the registry service must perform all the computation to identify the relevant publishers, to group these into equivalence classes, and then to generate the meta query plan and the query plan. This places a significant burden on the registry service which would conduct this for every consumer and republisher in the system. The registry service would also need to store all of the meta query plans in order to gain the maintenance benefit from generating them. However, the information that needs to be communicated to the agent is minimised as only contact details of the publishers in the plan together with the local query to be posed need be sent to the query agent.

In the second case, the computation that the registry service must perform is minimised. The registry service simply sends the details of all the publishers, including their view descriptions, that are registered in the system to the agent. However, this will generate an excess of network traffic as details of many publishers which are not

relevant to the continuous query will also be sent to the agent, i.e. details of publishers which can never be part of the query plan are also sent to the agent.

The third option provides a compromise between the previous two by minimising both the communication and computational load on the registry service. The registry service identifies which publishers are relevant to the continuous query, and only sends details of publishers that can potentially be used in the query plan but without performing all of the computation required to generate the meta query plan and query plan.

The situation is similar when there is a change in the publisher configuration, the query plans of existing continuous queries must be updated to reflect the new publisher configuration. Three cases, similar to those for a new continuous query, are considered:

1. The registry service identifies all continuous queries that are potentially affected by the change in the publisher configuration, updates each of the meta query plans and query plans accordingly and sends the updated plans to the agents.

2. The registry service informs all continuous queries of the change in the publisher configuration and the agent updates their own meta query plan and query plan accordingly.

3. The registry service identifies those continuous queries that are potentially affected by the change in the publisher configuration and informs their agents. Each of these agents then updates their meta query plan and query plan accordingly.

Again, the first case minimises the communication load on the registry service, the second case reduces the computational load on the registry service, and the third case attempts to minimise both the communication and computational load.

For both the query planning and the maintenance, following an approach that minimises both the communication and computational load on the registry service best meets the requirements of the motivating application domain of a Grid information and monitoring system and of distributed computing systems in general.

The query planning and maintenance mechanisms require that the stream integration system infrastructure provide certain capabilities. These are:

- Communication between components: the registry service needs to be able to send details of publishers to the consumer agent. These details will include how to contact the publisher agent and also the view condition of the publisher.

- Data streaming: the publisher agents must be able to stream data to the consumer and republisher agents that satisfy the query posed.

- Registration information: the publisher and consumer agents need to be able to register their details in the registry. The registry also needs a mechanism to eliminate details of components that no longer exist due to some network problem.

- Identification of relevant components: the registry service needs to be able to quickly find all relevant publishers for a consumer query and all relevant consumers and republishers for a change in the publisher configuration.

- Smart agents: the consumer agent must be able to generate meta query plans and query plans. This requires that the agent is capable of performing the subsumption check detailed in Section 5.3.2, and to group the republishers into equivalence classes.

The R-GMA system already provides much of the required infrastructure. Currently, due to the limited query planning employed for continuous queries it does not have smart agents. The next section will discuss the details of the R-GMA implementation and present its current query planning and maintenance techniques. The subsequent sections will then consider how the existing query planning and maintenance techniques can be extended to allow republishers to be used to answer a continuous query.

7.2 The R-GMA System

The implementation of the R-GMA system [89,108,111] began as part of the EU DataGrid project [80] and is continuing as part of the ongoing EU EGEE [81] project. During the EGEE project, the implementation of the R-GMA system has been undergoing a major redesign to consolidate the system and to make it more robust [10,111,114].

In order to isolate the development of the extended query planning and maintenance mechanisms from these changes in the implementation of the R-GMA system, a stable code release was chosen. This was R-GMA version 4.1.11 which formed part of the gLite 1.4 code release.

The R-GMA system has already taken implementation decisions on allocating query planning and maintenance tasks to specific components. The R-GMA system used the same criteria as discussed above to help ensure that the resulting system was a reliable, scalable distributed system that did not have a bottleneck.

The following sections will give a brief overview of the implementation of the R-GMA system and then discuss the continuous query planning and maintenance mechanisms. For the purposes of presentation, the terminology of the stream integration system will be used rather than the system names used in R-GMA. That is, the streams of data are published by producers, queries are posed by consumers, and republishers pose a query and make the resulting data available. In the R-GMA system these components are now known as primary producers, consumers, and secondary producers respectively.

7.2.1 Design of the Implemented R-GMA

The R-GMA implementation used consists of a set of services (offered as Java Servlets) that support the Grid resources (that play the role of producer or consumer) in publishing and querying the Grid information and monitoring data. The Grid resources interact with the services through application programming interfaces (APIs) offered in C, C++, Java, Perl, and Python.

A UML component diagram of the R-GMA services is presented in Figure 7.1. The dependencies in the diagram show which services make a request to another service. In a Grid deployment there would be several instances of the producer service and consumer service hosted on web servers across the Grid. Ideally, the registry service and schema service would also have several instances spread across the Grid. However, the version of R-GMA used did not support replication of the registry or schema services. This will not affect the implementation of the extended query planning and maintenance mechanisms.

The R-GMA services provide a realisation of the agent metaphor used in Section 4.1.

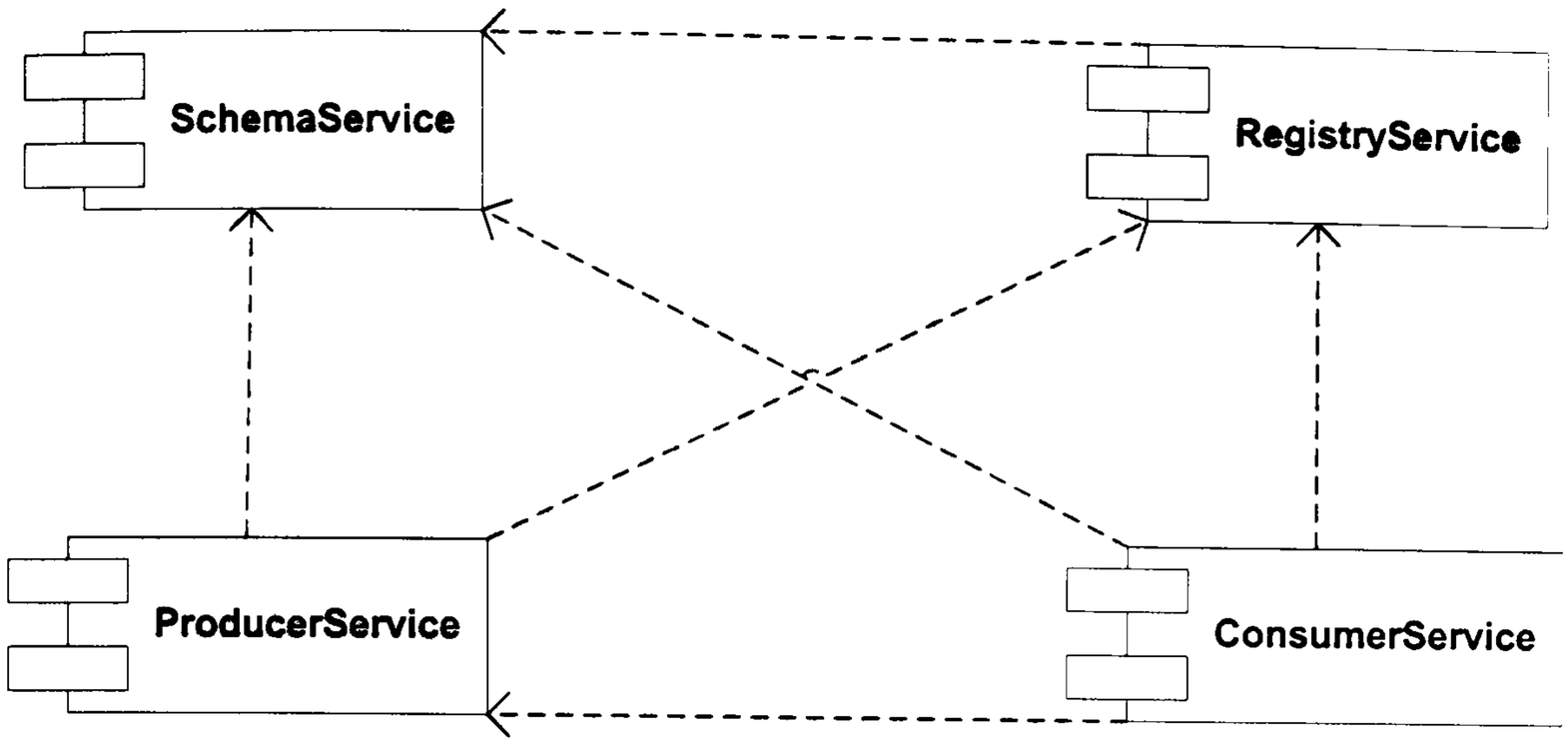


Figure 7.1: A UML component diagram of the services of the R-GMA system.

The agents provide all the R-GMA knowledge required to perform the tasks, e.g. the producer agent allows the producer to insert tuples and responds to queries on behalf of the producer. Each Grid resource has its own agent. A single service may host several agents.

The R-GMA system only provides limited republisher functionality. A republisher is able to pose one or more continuous queries which collect the data from several producers. The republisher makes either the history or the latest-state of the resulting streams available. This functionality is realised by combining several consumer instances together with a producer instance within a single component. At present, due to the query planning mechanisms employed in R-GMA, a republisher may not be used to answer a continuous query. However, these republishers are useful for answering one-time queries and make it possible for complex queries, such as a join, to be answered. Details of the query planning for one-time queries in the R-GMA system can be found in [108].

The schema and registry services are not directly accessible to the Grid resources. The functionality provided by these services is accessed through the agents. The schema maintains the set of relations in the global schema. It allows agents to look up properties of the relations, e.g. the attributes and types in a specific relation. Whenever a new query or view is declared, these must be checked for validity against the relations stored in the schema.

The registry service is responsible for three bits of functionality:

1. Storing details of the components that are registered with the system.

2. Identifying relevant producers for a query[1].

3. Identifying relevant consumers for a new producer.

Details of each of these will be discussed in the following sections.

7.2.2 Storing the Registration Data of the Components

The registry service maintains details of all publishers and consumers with continuous queries that are registered in the system. The registration details of the components need to be stored in a reliable data structure which can be searched easily and updated whenever a component is added to or removed from the system.

The details that need to be stored about the components must include the URL where the component's agent is located along with an identifier for the agent so that it can be contacted. The registry should also store details of the view registered by a producer and the continuous query posed by a consumer or a republisher.

In the R-GMA system, the view of a producer may only be a selection on a single relation in the global schema. The conditions in the view are limited to conjunctions of the form

$$\mathsf{attr} = \mathsf{value}. \tag{7.1}$$

Thus the registry service need only store the attribute-value combinations.

The continuous queries permitted are more expressive than the views of the producers but are still limited to selections on a single relation in the global schema. The query conditions are conjunctions of the form

$$\mathsf{attr\ op\ value}, \tag{7.2}$$

where the operator may be one of the following $\{<, \leq, =, \geq, >\}$[2]. Thus, for each conjunct the registry service needs to store the operator as well as the attribute and the value.

A database with a suitable schema meets these requirements as it can be searched efficiently and updated as required. A simplified version of the schema of the database

[1]Only continuous queries will be considered here.

[2]R-GMA does not support $\neq$.

Consumers(URL, connectionId, tableName, predicate, flags,
clientTimeStamp, terminationTime)
Producers(URL, connectionId, tableName, flags,
clientTimeStamp, terminationTime)
FixedStringColumns(URL, connectionId, tableName, columnName, value)
FixedIntColumns(URL, connectionId, tableName, columnName, value)
FixedRealColumns(URL, connectionId, tableName, columnName, value)

Figure 7.2: R-GMA registry database schema.

used by the registry service in R-GMA, which omits attributes introduced for the purposes of replicating the registry, is provided in Figure 7.2. The underlined attributes of each relation form the primary key.

The Consumers relation stores details of the continuous queries registered in the system. This includes the URL of the consumer service and the connectionId of the agent within that service. The tableName attribute stores the name of the relation that the consumer is querying and the predicate of the query is stored as a string in the predicate attribute. The flags attribute is used to encode whether the consumer is part of a republisher or not. The final two attributes, clientTimeStamp and terminationTime, are used to purge the database of old consumers. Together they are used as a form of soft state registration which allows the registry service to remove entries of defunct consumers, e.g. the consumer no longer exists due to a system crash and was unable to send a remove consumer message. In order that the consumer agent can maintain its entry in the registry database, it must periodically send a message to the registry service. Upon receiving this message, the registry service updates the values in these attributes. If this message fails to arrive, the value in the terminationTime attribute will be exceeded and the consumer is no longer considered to be available to the system.

Details of the producers are stored in the Producers relation. The attributes of this relation are the same as for the Consumers except that there is no predicate attribute.

This is because only the combination of attribute and value need to be stored, and this can be achieved using a separate set of relations: one for each of the data types supported by R-GMA. These relations are the FixedStringColumns, FixedIntColumns, and FixedRealColumns relations and are collectively referred to as the FixedColumns relations. The FixedColumns relations store the restricted value for each conjunct in the view that the producer declared with the value being stored in an appropriate type format. The producer may only restrict each attribute once.

To demonstrate the information stored in the registry, consider the following components.

The relations used in the example are the Network Throughput relation ntp and the Cluster Computing Element relation CECluster. As previously introduced in (4.1), the schema for ntp is

$$\mathsf{ntp}(\underline{\mathsf{from}}, \underline{\mathsf{to}}, \underline{\mathsf{tool}}, \underline{\mathsf{psize}}, \mathsf{latency}, [\mathsf{timestamp}]). \tag{7.3}$$

and as introduced in (4.6), the schema for CECluster is

$$\mathsf{CECluster}(\underline{\mathsf{clusterId}}, \mathsf{name}, \mathsf{URL}). \tag{7.4}$$

The details of the components are

- A consumer with the continuous query

$$\sigma_{\mathsf{tool}=\texttt{'ping'} \wedge \mathsf{psize}>24}(\mathsf{ntp}), \tag{7.5}$$

  which has an agent hosted on a suitable consumer service.

- Three producers with the view descriptions

$$S_1 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp}), \tag{7.6}$$

$$S_2 := \sigma_{\mathsf{from}=\texttt{'ral'} \wedge \mathsf{psize}=32}(\mathsf{ntp}), \tag{7.7}$$

$$S_3 := \sigma_{\mathsf{clusterId}=\texttt{'hw'}}(\mathsf{CECluster}), \tag{7.8}$$

  where each has an agent hosted on a suitable producer service.

- A republisher with the query

$$\sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{latency}=30}(\mathsf{ntp}), \tag{7.9}$$

  which has suitable consumer and producer agents hosted on appropriate services.

URL	connectionId	tableName	predicate	flags
con.info/ConsumerService	874	ntp	tool = 'ping' AND psize > 24	17
hw.ac.uk/ConsumerService	98	ntp	from = 'hw' AND latency = 30	25

(a) An instance of the Consumers relation.

URL	connectionId	tableName	flags
hw.ac.uk/ProducerService	38	CECluster	1
hw.ac.uk/ProducerService	42	ntp	1
ral.ac.uk/ProducerService	74	ntp	1
hw.ac.uk/ProducerService	45	ntp	12

(b) An instance of the Producers relation.

URL	connectionId	tableName	columnName	value
hw.ac.uk/ProducerService	38	CECluster	clusterId	hw
hw.ac.uk/ProducerService	42	ntp	from	hw
hw.ac.uk/ProducerService	42	ntp	tool	ping
ral.ac.uk/ProducerService	74	ntp	from	ral
hw.ac.uk/ProducerService	45	ntp	from	hw

(c) An instance of the FixedStringColumns relation.

URL	connectionId	tableName	columnName	value
ral.ac.uk/ProducerService	74	ntp	psize	32

(d) An instance of the FixedIntColumns relation.

URL	connectionId	tableName	columnName	value
hw.ac.uk/ProducerService	45	ntp	latency	30

(e) An instance of the FixedRealColumns relation.

Table 7.1: An instance of the database used by the registry service in R-GMA.

The registration information that is stored for these components is shown in Table 7.1. The termination periods and last contact time have been omitted as they are not important for the discussion.

Note that the value of the flags attribute is a combination value. That is, there are values that correspond to certain types of query support, or component type, which can be added together to provide support for multiple query types. These base values are provided in Table 7.2. As an example, consider the flag value of the republisher in the Producers relation which was given as 12. This is interpreted as the producer is part of a republisher (8) and supports history queries (4). These values added together total 12.

The next sections will consider how the information stored by the registry service is used to generate query plans for continuous queries and how these are maintained.

Flag	Meaning
1	Continuous
2	Latest
4	History
8	Republisher
16	Consumer

Table 7.2: The base values and their interpretation for the flags attribute.

7.2.3 Continuous Query Planning

In the R-GMA system, a plan for a continuous query will only consider producers that are not part of a republisher. This simplifies the query planning problem since there is no redundancy in the data considered and there is no mechanism by which a loop in the information path as detailed in Section 6.2 can occur. The fact that republishers cannot be used also means that the query plans that are generated are simpler than those described in Section 5.3 because there is no longer a choice in where to retrieve information. The query plan only needs to ensure that all producers that are relevant for the query are used to retrieve information.

Hence, the query planning tasks are

1. Identify all relevant producers for a continuous query.

2. Start streaming data from all the relevant producers.

The first of these tasks is conducted by the registry service upon receiving the registration of a new continuous query. The registry service will then pass the details of all the relevant producers to the agent for the continuous query which will then contact all the relevant producers with a start streaming message.

When a new continuous query is registered, it contacts the registry service to inform it of its query over a single relation in the global schema and to retrieve the producers that are relevant for the query. The registry service must search the stored registration information about the components to identify those that are relevant. This can be achieved by querying the database where the details are stored.

A producer is relevant for a continuous query if both of the following are true:

1. It publishes data for the global relation mentioned in the continuous query.

2. The view condition of the producer and the query condition are satisfiable.

These conditions can be checked by a query to the registry database which identifies all the producers which publish for the global relation and eliminates those which have a contradictory view condition. A producer has a contradictory view condition if one of the conjuncts in the view sets an attribute to a value that does not satisfy the query. Since all the queries are conjuncts of attributes, it is sufficient to find one attribute for which the view is contradictory.

Consider the consumer registering the global query

```
SELECT *
FROM   ntp
WHERE  from = 'hw' AND
       tool = 'ping' AND
       psize >= 32.
```
(7.10)

A producer would not be relevant if

- It published for a relation other than ntp. For example, producer 38 at hw.ac.uk publishes for the CECluster relation.
- It has the from attribute set to a value other than `'hw'`. For example, producer 74 at ral.ac.uk has this attribute set to `'ral'`.
- It has the tool attribute set to a value other than `'ping'`.
- It has the psize attribute set to a value less than 32.

These conditions can be tested by a SQL query to the registry service database, Query 7.11. The outer query identifies all those producers which publish for the relation ntp. The sub-queries identify those producers which have a conjunct of a specific type that does not satisfy the query condition. This is done for each of the conditions in the global continuous query.

```
SELECT p.URL, p.connectionId, p.flags
FROM   Producers p
WHERE p.tableName = 'ntp' AND
NOT EXISTS (SELECT *
                FROM   FixedStringColumns s
                WHERE p.URL = s.URL AND
                      p.connectionId = s.connectionId AND
                      p.tableName = s.tableName AND
                      ((s.columnName = 'from' AND
                      s.value <> 'hw') OR
                      (s.columnName = 'tool' AND
                      s.value <> 'ping')))
AND
NOT EXISTS (SELECT *
                FROM   FixedIntColumns i
                WHERE p.URL = i.URL AND
                      p.connectionId = i.connectionId AND
                      p.tableName = i.tableName AND
                      ((i.columnName = 'psize' AND
                      i.value < 32)))
```
(7.11)

The result set of this query contains all the producer agents that are relevant for the consumer query, including those producer agents which represent a republisher. For the instance of the registry service database given in Table 7.1, the result is given in Table 7.3. This result set contains details of producers that are part of a republisher. Therefore, before returning the list of relevant producers to the consumer agent, the flags are used to remove those producers that are part of a republisher. For the example, the second entry refers to a republisher which would then be eliminated from the result set.

When the agent of the continuous query receives the list of relevant producers, it contacts all of them. It passes the query condition as a parameter to the start streaming message sent to the producer agent. The producer agent, on receiving a start streaming message, then sends those tuples which satisfy the condition of the

URL	connectionId	flags
hw.ac.uk/ProducerService	42	1
hw.ac.uk/ProducerService	45	12

Table 7.3: The result of executing Query (7.11) to the registry database instance in Table 7.1.

query to the agent of the continuous query and continues to send tuples whenever a satisfying tuple is inserted.

7.2.4 Maintaining Continuous Query Plans

The task of maintaining query plans can be broken down into two stages. The first stage is to detect when the publisher configuration has changed. Once a change has been detected, the continuous queries that are potentially affected must be identified and informed of the change.

In the R-GMA implementation, there are two cases where a change in the publisher configuration affects the query plan of a continuous query. These are:

1. Adding a new producer.

2. Removing an existing producer.

In the first case, the detection of a new producer is straightforward as the agent of the new producer must inform the registry service of the existence of the producer. The second case is more difficult. The system cannot simply rely on the agent of the producer informing the registry service that it is going to stop. Since R-GMA is a distributed system, a producer could fail for a number of reasons and be unable to send such a message. R-GMA maintains the list of active producers through the terminationTime attribute in the Producers relation. The agent of each producer is expected to periodically send a message to the registry service to update their terminationTime value so that it is always set to a time in the future. If a producer agent fails to update the terminationTime value, this results in the value becoming a time in the past. Before any operation is performed by the registry service, producers that have exceeded their terminationTime, i.e. those which have a value in the past, are

removed. Thus, the registry service is able to detect any changes to the publisher configuration.

Note, there is no need to maintain the query plan of a continuous query when a republisher is added to or removed from the system as these cannot be used to answer a continuous query in the current R-GMA system.

Once a change in the publisher configuration has been detected, the registry service must then identify those continuous queries for which the change has an effect. For simplicity, the following shall discuss the case when a producer is added to the system. The same mechanism applies when a producer is no longer in the system. It is worth noting that when a producer is added the affected queries *must* be informed of the change as otherwise their query plans will no longer return the complete answer. However, in the case that a producer is no longer available the affected queries are informed in order that they may keep their query plans up to date.

When a new producer is added, the registry service must identify those queries for which the producer is relevant. This again involves a satisfiability test between the condition in the view of the producer and the conditions of the continuous queries. However, it is now the query conditions that are stored by the registry service and the view condition that is passed in.

Since the query conditions are not stored in a structured manner, the query to the registry service database cannot perform the satisfiability test. The test for finding relevant producers relied on the structured format of the view conditions. Thus, the R-GMA registry service poses a query to its database to identify those queries that consume for the global relation mentioned in the view of the producer. The registry service must then parse the condition of each of the continuous queries and perform a satisfiability test. Again, the view condition and the query condition are satisfiable if it is possible to make them both true, i.e. if they do not have a contradictory condition.

Consider the addition of the producer with the view condition

$$\sigma_{\mathsf{from}=\texttt{'ral'} \wedge \mathsf{tool}=\texttt{'ping'}}(\mathsf{ntp}), \tag{7.12}$$

which publishes data where the measurements originate at `'ral'` and are measured with the `'ping'` tool for the ntp global relation. The query that the registry service

URL	connectionId	predicate	flags
con.info/ConsumerService	874	tool = 'ping' AND psize > 24	17
hw.ac.uk/ConsumerService	98	from = 'hw' AND latency = 30	25

Table 7.4: The result of executing Query (7.13) to the registry database instance in Table 7.1.

would pose to its database is

$$\begin{aligned} &\text{SELECT URL, connectionId, predicate, flags} \\ &\text{FROM Consumers} \\ &\text{WHERE tableName} = \text{'ntp'}. \end{aligned} \tag{7.13}$$

The result produced when Query (7.13) is posed to the registry database instance given in Table 7.1 is given in Table 7.4. The query conditions of the consumers returned would then be parsed and a satisfiability test performed on each conjunct to identify if the new producer is relevant. The first of the consumers passes the satisfiability test whereas the second one fails due to the contradiction in the values for the from attribute. The consumer agent 874 at con.info/ConsumerService would then be contacted with the details of the new producer.

When an agent for a continuous query receives a notification from the registry service of a new relevant producer, it will contact the producer's agent with its query so that it can start streaming tuples from the producer. There is no need to check whether the tuples can arrive at the consumer from another source as republishers cannot be used to answer a continuous query and the streams of the producers are assumed to be disjoint. Thus, all relevant producers should be added to the execution plan for the continuous query. In the case that a producer is being removed, the producer is removed from the list of active connections for the query.

7.3 Improving the Query Planning and Plan Maintenance Mechanisms

It has already been argued that the R-GMA system would provide a suitable basis for implementing the query planning and maintenance mechanisms developed in this

thesis. The previous section gave an overview of R-GMA version 4.1.11, with specific details of how the registration information about components is stored in the registry service and the query planning and maintenance mechanisms that exist for a continuous query. This section will now detail the extensions that were required to implement the new query planning and maintenance mechanisms.

The main difference between the existing mechanisms and the new mechanisms is that republishers can now be used to answer a continuous query. The consequences of this are:

- Republishers require symmetry between the treatment of producers and consumers in the registry service.
- The relevance tests are complicated by the introduction of entailment of measurement attributes as detailed in Section 5.3.
- Query plans are more complicated as a choice must be made as to which publisher to contact for each part of the answer stream as discussed in Section 5.3.
- plan maintenance is more complicated as the cases where a republisher is added to or removed from the publisher configuration must be handled as shown in Section 6.1.

These points will now be considered in the following sections.

7.3.1 Storing the Registration Data of the Components

In Section 7.2.2, details of how the R-GMA system currently stores the registration details of consumers, producers, and republishers was presented. The language that should be supported for a republisher query, and hence the view condition of their producer, are conjunctions of conditions of the form

$$\mathsf{attr\ op\ value}, \tag{7.14}$$

where op is one of $\{\leq, <, =, >, \geq\}$[3]. However, the R-GMA system is currently unable to represent the view condition registered by the producer component of the

[3]Again, the operator $\neq$ is not supported as this is not supported in the current R-GMA system.

republisher since producer views are limited to conjunctions of conditions of the form

$$\mathsf{attr} \;=\; \mathsf{value}. \tag{7.15}$$

Two approaches to storing the registration information were considered. The first was to follow the approach used by the registry service to handle continuous queries, i.e. to store the query as a string in the database. This requires that each lookup to the database to retrieve details of relevant publishers, or queries, needs to parse the string containing the view conditions and then perform the relevance test in the registry service code. The second approach was to develop a suitable registry service database schema by which the queries of the consumers, producers, and republishers, can be stored in a structured manner. This approach would allow the relevance test to be performed as part of the query to retrieve the registration information.

Implementations of both approaches with the functionality of the current R-GMA system were developed to allow experiments to be conducted into the performance of each. Details of the tests, together with the results, are provided in Section 8.1. The results showed that there was a significant performance gain to the registry service in adopting the second approach whereby the query conditions are stored in a structured manner.

The database schema for storing continuous queries in a structured manner is presented in Figure 7.3. A side effect of the design is that the expressivity allowed for the views of the producers is increased to that of the continuous query.

In the new schema, the Producers and Consumers relations of the R-GMA registry service database have been merged into one common Components relation. This is because the same information is being stored about consumers, producers, and republishers. An additional attribute, componentType, was introduced to help distinguish between the consumers, producers, and republishers, although this could also be achieved with the flags attribute. The FixedColumns relations are now used to store details of both the view conditions of the publishers and the query conditions of the continuous queries.

In order to allow ranges of numeric types additional attributes were needed in the FixedIntColumns and FixedRealColumns relations. These attributes store, for each bound, the type of the bound (i.e. infinite, inclusive, or exclusive) and the value of the

Components(URL, connectionId, tableName, flags, componentType,
clientTimeStamp, terminationTime)
FixedStringColumns(URL, connectionId, tableName, columnName, value)
FixedIntColumns(URL, connectionId, tableName, columnName, lowerBoundType,
lowerBoundValue, upperBoundValue, upperBoundType)
FixedRealColumns(URL, connectionId, tableName, columnName, lowerBoundType,
lowerBoundValue, upperBoundValue, upperBoundType)

Figure 7.3: Improved registry database schema.

bound. For example, consider the following consumer, producers, and republishers that are similar to those presented in Section 7.2.2 but which exploit the fact that producers, and the producers of republishers, can now declare more expressive views.

- A consumer with the continuous query

$$\sigma_{\mathsf{tool}=\texttt{'ping'} \wedge \mathsf{psize}>24 \wedge \mathsf{psize}\leq 128}(ntp), \tag{7.16}$$

  which has an agent hosted on a suitable consumer service.

- Two producers with the view descriptions

$$S_1 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{tool}=\texttt{'ping'}}(ntp), \tag{7.17}$$

$$S_2 := \sigma_{\mathsf{from}=\texttt{'ral'} \wedge \mathsf{psize}>=32}(ntp), \tag{7.18}$$

  where each has an agent hosted on a suitable producer service.

- The republishers with the queries

$$R_1 := \sigma_{\mathsf{from}=\texttt{'hw'} \wedge \mathsf{latency}<30}(ntp), \tag{7.19}$$

$$R_2 := \sigma_{\mathsf{latency}<30}(ntp), \tag{7.20}$$

  which both have suitable consumer and producer agents hosted on appropriate services.

URL	connectionId	tableName	flags	componentType
hw.ac.uk/ProducerService	42	ntp	1	Producer
ral.ac.uk/ProducerService	74	ntp	1	Producer
hw.ac.uk/ProducerService	45	ntp	13	Producer
con.info/ConsumerService	874	ntp	17	Consumer
hw.ac.uk/ConsumerService	98	ntp	25	Consumer
rep.info/ConsumerService	23	ntp	25	Consumer
rep.info/ProducerService	52	ntp	13	Producer

(a) An instance of the Components relation.

URL	connectionId	tableName	columnName	value
hw.ac.uk/ProducerService	42	ntp	from	hw
hw.ac.uk/ProducerService	42	ntp	tool	ping
ral.ac.uk/ProducerService	74	ntp	from	ral
hw.ac.uk/ProducerService	45	ntp	from	hw
con.info/ConsumerService	874	ntp	tool	ping
hw.ac.uk/ConsumerService	98	ntp	from	hw

(b) An instance of the FixedStringColumns relation.

URL	connectionId	tableName	columnName	LBT	LBV	UBV	UBT
ral.ac.uk/ProducerService	74	ntp	psize	inc	32	null	inf
con.info/ConsumerService	874	ntp	psize	non	24	128	inc

(c) An instance of the FixedIntColumns relation.

URL	connectionId	tableName	columnName	LBT	LBV	UBV	UBT
hw.ac.uk/ProducerService	45	ntp	latency	inf	null	30	non
hw.ac.uk/ConsumerService	98	ntp	latency	inf	null	30	non
rep.info/ConsumerService	23	ntp	latency	inf	null	30	non
rep.info/ProducerService	52	ntp	latency	inf	null	30	non

(d) An instance of the FixedRealColumns relation.

Table 7.5: An instance of the database used by the improved registry service.

The registration information that is stored for these components is shown in Table 7.5. For the purposes of presentation the abbreviations in Table 7.6 have been used. Again, the termination periods and last contact time have been omitted as they are not important for the discussion. The database design could also be easily extended to allow ranges of string values.

To illustrate how the improved registry service database is populated consider the consumer with the query given in (7.16). The details of the consumer's agent are stored in the Components relation, shown as the fourth row in Table 7.5(a). The condition of the consumer's query consists of three conjuncts which restrict the values of two attributes. The tool attribute, which is a string, is restricted to the value `'ping'`. This is shown as the fifth row of the FixedStringColumns relation in Table 7.5(b).

Bound type	Meaning
LBT	lowerBoundType
LBV	lowerBoundValue
UBV	upperBoundValue
UBT	upperBoundType

(a) Boundary condition

Range type	Meaning
inf	infinite bound
inc	inclusive bound
non	exclusive bound

(b) Type of boundary condition

Table 7.6: Abbreviations used in the presentation of the improved registry database instance.

The two conjuncts referring to the integer attribute psize restrict the values of this attribute to the range

$$(24, 128], \tag{7.21}$$

which has an exclusive lower bound and an inclusive upper bound. This is shown as the second row of the FixedIntColumns relation in Table 7.5(c).

The publishers are treated in the same manner and give rise to the other rows of the registry instance displayed in Table 7.5. Note that the republishers are each represented by two entries in the Components relation, one for the producer part and one for the consumer part. This also results in two entries in the FixedColumns relations as appropriate. This approach was adopted in order to minimise the alterations to the services and agents of the R-GMA system, although with the new database design the duplication is not needed.

7.3.2 Finding Relevant Publishers

In the current R-GMA system, the test for finding relevant publishers for a continuous query only considers producers. The test is performed by a query to the registry service database and consists of ensuring that the view of a producer and the query condition are satisfiable. However, when republishers can be used to answer a continuous query the relevance criteria become more complex. Since there is a difference between the relevance criteria for a consumer query and those for a republisher query (see Section 6.2), the following discussion will first concentrate on a consumer query before presenting the approach adopted for a republisher query.

As stated in Lemma 5.10, a publisher P with view condition $D^\kappa \wedge D^\mu$ is relevant for a consumer query with condition $C^\kappa \wedge C^\mu$ if both of the following hold:

1. $C \wedge D$ is satisfiable.

2. $C^\mu \models D^\mu$.

The first criterion ensures that the publisher can potentially contribute tuples to the answer stream of the query while the second ensures the weak order property of the answer stream.

The first criterion requires that a satisfiability test is performed. Since the conditions in the view condition can now include ranges, the satisfiability test is more complex than that performed in the R-GMA system. However, the same principle can be applied. Namely, the view of the publisher and the query are not satisfiable if any one of their conditions contradict, i.e. they cannot be made true by the same tuple. For the numeric types, this means that the ranges registered for the view do not overlap with the range required for the query. The criteria for when two intervals are contradictory are given in Table 7.7. It is straightforward to extend the previous query to the registry service database to cover this case, although care must be taken to consider the data type (i.e. whether it is an integer or a real), the range boundary type (i.e. whether it is infinite, inclusive, or exclusive), and the value.

The second criterion for relevance requires an entailment test on the measurement attributes of the global relation. The view condition of a publisher fails the entailment test if either of the following are true:

- It restricts a measurement attribute that the query does not.
- It has a more restrictive condition on a measurement attribute than the condition in the query, e.g. the query requires a measurement attribute to be less than 50 but the publisher restricts the attribute to be less than 30.

This test can also be conducted as a sub-query to the registry service database that eliminates those publishers with a view condition on a measurement attribute that is more restrictive than the query condition. This will be demonstrated through examples.

	$[a,b]$	$(a,b]$	$[a,b)$	(a,b)	$(-\infty,b]$	$(-\infty,b)$	$[a,\infty)$	(a,∞)	$(-\infty,\infty)$
$[c,d]$	$d<a$ $\vee$ $b<c$	$d\leq a$ $\vee$ $b<c$	$d<a$ $\vee$ $b\leq c$	$d\leq a$ $\vee$ $b\leq c$	$b<c$	$b\leq c$	$d<a$	$d\leq a$	
$(c,d]$	$d<a$ $\vee$ $b\leq c$	$d\leq a$ $\vee$ $b\leq c$	$d<a$ $\vee$ $b\leq c$	$d\leq a$ $\vee$ $b\leq c$	$b\leq c$	$b\leq c$	$d<a$	$d\leq a$	
$[c,d)$	$d\leq a$ $\vee$ $b<c$	$d\leq a$ $\vee$ $b<c$	$d\leq a$ $\vee$ $b\leq c$	$d\leq a$ $\vee$ $b\leq c$	$b<c$	$b\leq c$	$d\leq a$	$d\leq a$	
(c,d)	$d\leq a$ $\vee$ $b\leq c$	$d\leq a$ $\vee$ $b\leq c$	$d\leq a$ $\vee$ $b\leq c$	$d\leq a$ $\vee$ $b\leq c$	$b\leq c$	$b\leq c$	$d\leq a$	$d\leq a$	
$(-\infty,d]$	$d<a$	$d\leq a$	$d<a$	$d\leq a$			$d<a$	$d\leq a$	
$(-\infty,d)$	$d\leq a$	$d\leq a$	$d\leq a$	$d\leq a$			$d\leq a$	$d\leq a$	
$[c,\infty)$	$b<c$	$b<c$	$b\leq c$	$b\leq c$	$b<c$	$b\leq c$			
(c,∞)	$b\leq c$	$b\leq c$	$b\leq c$	$b\leq c$	$b\leq c$	$b\leq c$			
$(-\infty,\infty)$									

Table 7.7: Boundary and value criteria for when an interval a, b contradicts an interval c, d. An inclusive boundary condition is represented with a square bracket while an exclusive boundary condition is represented with a round bracket.

```
SELECT c.URL, c.connectionId, c.flags
FROM   Components c
WHERE c.componentType = 'Producer' AND
          c.tableName = 'ntp' AND
NOT EXISTS (SELECT *
            FROM   FixedStringColumns s
            WHERE c.URL = s.URL AND
                  c.connectionId = s.connectionId AND
                  c.tableName = s.tableName AND
                  ((s.columnName = 'from' AND
                  s.value <> 'hw') OR
                  (s.columnName = 'tool' AND
                  s.value <> 'ping')))
AND
NOT EXISTS (SELECT *
            FROM   FixedIntColumns i
            WHERE c.URL = i.URL AND
                  c.connectionId = i.connectionId AND
                  c.tableName = i.tableName AND
                  (i.columnName = 'psize' AND
                  ((i.upperBoundType = 'inclusive' AND
                  i.upperBoundValue < 32) OR
                  (i.upperBoundType = 'exclusive' AND
                  i.upperBoundValue <= 32))))
AND
NOT EXIST (SELECT *
                  FROM   FixedRealColumns r
                  WHERE c.URL = r.URL AND
                        c.connectionId = r.connectionId AND
                        c.tableName = r.tableName AND
                        (r.columnName = 'latency'))
```
(7.22)

URL	connectionId	flags
hw.ac.uk/ProducerService	42	1

Table 7.8: The result of executing Query (7.22) to the registry database instance in Table 7.5.

Consider again the global Query (7.10) on Page 111. The query generated to retrieve the relevant publishers from the new registry service database is given in Query (7.22) on Page 123.

The first two sub-queries perform the satisfiability test with the integer test being extended to check for the ranges of the values using the criteria from Table 7.7. Since the condition in the query is of the form $[a, \infty)$ the eighth column gives the criteria that the database query must check. The range values stored in the database contradict the query condition if one of the following is true:

1. The stored upper bound type is inclusive and the value of the bound is less than 32.
2. The stored upper bound type is exclusive and the value of the bound is less than or equal to 32.

Note that the first of these has a less than test while the second has a less than or equal to test. This is because the boundary conditions 31] and 32) are equivalent for integers and the satisfiability test must ensure that the stored upper bound is less than or equivalent to 31].

The third sub-query of Query (7.22) ensures that the measurement entailment for the single measurement attribute of ntp holds.

The result of posing Query (7.22) to the registry service database instance given in Table 7.5 is shown in Table 7.8. The republishers that exist in the system are not returned as they restrict the value of the measurement attribute latency.

If Query (7.22) had the additional condition that the latency must be less than 10

seconds, i.e. it was the query

```
SELECT *
FROM   ntp
WHERE  from = 'hw' AND
       tool = 'ping' AND
       psize >= 32 AND
       latency< 10,
```
(7.23)

then the republishers would also be returned as relevant publishers as the query has a more restrictive condition on the measurement attribute than the republishers do. The query that would be posed to the registry service database in this case is given in Query (7.24) on Page 126.

The first two sub-queries of Query (7.22) would appear in Query (7.24) but have been replaced by dots for the purposes of presentation. The third sub-query of Query (7.22) has been replaced by the two sub-queries shown in Query (7.24).

The first of the sub-queries shown in Query (7.24) performs the satisfiability test (Criterion 1 of the relevance test) applying the criteria of the seventh column of Table 7.7. Since latency is of type Real and for the Reals the two boundary conditions [x and (x are essentially equivalent due to the infinite nature of the domain, the test criteria is with the value 10. However, if latency had been an integer then the test conditions would have been:

1. The stored lower bound type is inclusive and the value of the bound is greater than or equal to 10.

2. The stored lower bound type is exclusive and the value of the bound is greater than or equal to 9.

The second of the sub-queries shown in Query (7.24) performs the entailment test (Criterion 2 of the relevance test). This ensures that all of the measurement values that the query requests can be provided by the publishers returned by the database query. A publisher does not provide all of the measurement values for Query (7.23) if one of the following holds:

1. The stored lower bound type is not infinite.

```
SELECT c.URL, c.connectionId, c.flags
FROM   Components c
WHERE c.componentType = 'Producer' AND
        c.tableName = 'ntp' AND
⋮
AND
NOT EXISTS (SELECT *
              FROM   FixedRealColumns r
              WHERE c.URL = r.URL AND
                    c.connectionId = r.connectionId AND
                    c.tableName = r.tableName AND
                    (r.columnName = 'latency' AND
                    ((r.lowerBoundType = 'inclusive' AND
                    r.lowerBoundValue >= 10) OR
                    (r.lowerBoundType = 'exclusive' AND
                    r.lowerBoundValue >= 10))))
AND
NOT EXISTS (SELECT *
              FROM   FixedRealColumns r
              WHERE c.URL = r.URL
                    c.connectionId = r.connectionId AND
                    c.tableName = r.tableName AND
                    (r.columnName = 'latency' AND
                    ((r.lowerBoundType <> 'infinite') OR
                    (r.upperBoundType = 'inclusive' AND
                    r.upperBoundValue < 10) OR
                    (r.upperBoundType = 'exclusive' AND
                    r.upperBoundValue < 10))))
```
(7.24)

2. The stored upper bound type is inclusive and the value is less than 10.

3. The stored upper bound type is exclusive and the value is less than 10.

The first test ensures that the lower bound of the consumer query is met while the last two ensure the upper bound condition of the query.

Finally, the case where the continuous query, for which relevant publishers are being sought, is part of a republisher shall be considered. In this case, as discussed in Section 6.2, the definition of when a publisher is relevant for the query should be replaced with that for strong relevance given in Proposition 6.5. That is, a republisher is strongly relevant for the query if it is strictly subsumed by the query. This subsumption test is conducted by the registry service code once all the relevant publishers have been identified using the queries shown above. The approach used for the subsumption test shall be discussed in the next section as it is central to the way in which meta query plans and query plans are generated.

7.3.3 Constructing the Query Plan

The current R-GMA system has only a rudimentary query plan in that it contacts all of the relevant producers. When republishers are allowed to be used to answer a continuous query, a choice must be made in the query plan as to which publishers to use. Section 5.3 presented a mechanism for constructing a meta query plan for a continuous query and using the meta query plan to derive a query plan. The same method can be applied for both consumer and republisher queries since it is the relevance criteria used that is important when planning a continuous query that is part of a republisher. For the purposes of the discussion, the following will consider generating a query plan for a consumer query.

In the R-GMA system, when a consumer agent registers its continuous query it receives from the registry service a list of details about relevant producers. The details returned about each producer are the URL and the connection identifier for the producer agent along with the value for the flags attribute. However, in order to construct the meta query plans detailed in Section 5.3, the consumer agent needs to be able to reason about the view conditions registered by the publishers. To facilitate this, the interface between the registry service and the consumer agent was extended

so that, for each of the relevant publishers, the consumer agent also receives details of the view condition registered in a structured format.

To construct the meta query plan for a query, the consumer agent must identify those publishers which are maximal relevant for the query and group the maximal relevant republishers into equivalence classes. The agent first considers the relevant republishers and constructs the equivalence classes of the maximal relevant republishers. The algorithm used to construct the equivalence classes is given in Figure 7.4.

The algorithm takes as its inputs a list containing the details of the relevant republishers and the consumer's continuous query. The result of the algorithm is a set of equivalence classes which contain the maximal relevant republishers for the given query in the current publisher configuration. The algorithm assumes the existence of certain methods.

getRepresentative(e): This method takes an equivalence class as its input and returns the details of one of the republishers in the class.

subsumedWRT(R_1, q, R_2): This method takes a republisher description R_1, a continuous query q, and another republisher description R_2 as its inputs. It returns true if it holds that

$$R_1 \preceq_q R_2 \tag{7.25}$$

removeSubsumedClasses($R, q,$ eClasses): This method takes a republisher, a query, and a set of equivalence classes as its input. It removes from the set of equivalence classes those that are subsumed with respect to the query by the republisher.

eClass(R): This method takes a republisher as its input and returns an equivalence class consisting of the republisher.

The algorithm itself iterates over the set of relevant republishers and groups them into equivalence classes containing only the maximal relevant republishers. For each republisher R in the set of relevant republishers $\mathbf{R}$ there are four cases:

1. Republisher R is equivalent to a republisher R' that is a member of an equivalence class e. This is the test conducted on line 7 of the algorithm. The

```
Input: q a continuous query
R the set of relevant republishers for q
eClasses = ∅
for all R ∈ R do
    found = false
    for all e ∈ eClasses do
        R' = getRepresentative(e)
        if subsumedWRT(R, q, R') and subsumedWRT(R', q, R) then
            e ∪ { R }
            found = true
            break for
        else if subsumedWRT(R, q, R') then
            found = true
            break for
        else if subsumedWRT(R', q, R) then
            eClasses \ { e }
            removeSubsumedClasses(R, q, eClasses)
            e' = eClass(R)
            eClasses ∪ { e' }
            found = true
            break for
        end if
    end for
    if found == false then
        e' = eClass(R)
        eClasses ∪ { e' }
    end if
end for
return eClasses
```

Figure 7.4: Algorithm to generate equivalence classes of republishers.

republisher R is added to the equivalence class e, line 8. There is no need to compare R with any of the other equivalence classes as it can only be equivalent to one such class.

2. Republisher R is strictly subsumed by some republisher R' in an equivalence class. This is the test conducted on line 11 of the algorithm. This implies that R is not maximal relevant for q in the current publisher configuration. No further comparisons are needed for republisher R.

3. Republisher R strictly subsumes some republisher R' which has been put into an equivalence class e. This is the test conducted on line 14 of the algorithm. This means that R' and the other members of the equivalence class e are not maximal relevant for q in the current publisher configuration. The equivalence class e should be removed from the set of equivalence classes. Additionally, it may be possible that R strictly subsumes w.r.t. q some of the other equivalence classes that have been generated. These are removed with the call to removeSubsumedClasses on line 16. A new equivalence class is then constructed for R and added to the set of equivalence classes, lines 17-18.

4. Republisher R is not comparable with any of the existing equivalence classes. This case is captured in line 23 of the algorithm. In this case, republisher R is maximal relevant for q with respect to the relevant republishers so far considered. A new equivalence class is formed and added to the set of equivalence classes, lines 24-25.

Note that for the first republisher considered there will not be any existing equivalence classes and so a new equivalence class is created. It may subsequently turn out that this republisher is not in fact maximal relevant for q in the current publisher configuration. In this case the equivalence class would be removed by some subsequent republisher matching case 3.

At the heart of the algorithm to generate the set of equivalence classes for a query is the subsumption test with respect to a query. As stated in Section 5.3.2, a publisher P with view condition $\sigma_{D^\kappa \wedge D^\mu}(r)$ is subsumed with respect to a query q with condition $\sigma_{C^\kappa \wedge C^\mu}(r)$ by a republisher R with view condition $\sigma_{E^\kappa \wedge E^\mu}(r)$, written $P \preceq_q R$, if

and only if

$$D^{\kappa} \wedge C^{\kappa} \models E^{\kappa}. \tag{7.26}$$

For the conditions considered, the entailment (7.26) holds if for each conjunct in E^{κ} either D^{κ} or C^{κ} has an equal or more restrictive condition. This can be tested by considering each conjunct individually. For example, republisher R_1 defined in (7.19) subsumes with respect to Query (7.23) republisher R_2 defined in (7.20) since the query restricts the attribute tool to the value `'ping'`and the attribute from to the value `'hw'`. In fact, the republishers are equivalent with respect to the query.

Once the equivalence classes have been constructed they are used to identify the maximal relevant producers for the query. A producer is maximal relevant if there does not exist a republisher in one of the equivalence classes that subsumes with respect to the query the producer. The same subsumption test is used, using a republisher to represent each equivalence class.

It is straightforward to implement the rest of the query planning mechanisms described in Section 5.3.

7.3.4 Improving the Plan Maintenance

In the R-GMA system, the plan maintenance consisted of:

1. Detecting when there was a change in the producers in the system.

2. Identifying the continuous queries that are potentially affected by the change in the producers.

3. For each affected continuous query, changing the set of active connections.

Since the information stored about the publishers and the query plans are more complex in the extended system, the techniques for maintaining query plans need to be altered.

The mechanisms implemented in R-GMA for detecting a change in the publisher configuration, see Section 7.2.4, are general. They will allow the registry service to detect whenever there is a change in the publisher configuration. i.e. the registry service will detect whenever a producer or republisher is added to, or removed from,

the system. However, the extended system is required to perform maintenance operations when a republisher is added or removed as well as when a producer is added or removed.

When the registry service detects a change in the publisher configuration, it must identify those continuous queries for which the publisher is relevant. This can be done in a similar way as when identifying the relevant publishers for a new query since the registration information about continuous queries is now stored in the same way as for publishers. However, in this case it is the continuous queries that are stored in the registry service database and the publisher that is passed in as the parameter.

The first criterion of the relevance test can be performed in the same way, i.e. the same satisfiability test on the conditions is performed. However, the entailment test that checks the second criterion for relevance needs to be altered. Note that this test only needs to be performed if the publisher is a republisher. This is because the producers are not permitted to restrict the measurement attributes and as such will always pass the entailment test since any condition entails true. For the case where the change is due to a republisher, the continuous query stored in the registry service database fails the test if either of the following is true:

1. It does not restrict a measurement attribute that is restricted in the republisher's query.

2. The restriction on a measurement attribute is more general than that in the republisher's query.

These can be tested as a sub-query to the registry service database.

For example, consider again the system configuration represented in Table 7.5 and suppose that republisher R_2 with the Query (7.20) on Page 118 is no longer available. The query generated to identify the affected continuous query agents from the registry service database is given in Query (7.27) on Page 133.

The first sub-query performs the satisfiability test to ensure that the continuous queries identified have an overlapping range for the latency attribute. The satisfiability test is the same as for when a new continuous query was registered and uses the criteria specified in Table 7.7. Since latency is of type real. the continuous queries in

```
SELECT c.URL, c.connectionId, c.flags
FROM   Components c
WHERE c.componentType = 'Consumer' AND
      c.tableName = 'ntp' AND
NOT EXISTS (SELECT *
            FROM   FixedRealColumns r
            WHERE c.URL = r.URL AND
                  c.connectionId = r.connectionId AND
                  c.tableName = r.tableName AND
                  (r.columnName = 'latency' AND
                  ((r.lowerBoundType = 'inclusive' AND
                  r.lowerBoundValue >= 30) OR
                  (r.lowerBoundType = 'exclusive' AND
                  r.lowerBoundValue >= 30))))
AND EXISTS (SELECT *
            FROM   FixedRealColumns r
            WHERE c.URL = r.URL
                  c.connectionId = r.connectionId AND
                  c.tableName = r.tableName AND
                  r.columnName = 'latency' AND
                  (r.lowerBoundType = 'infinite' AND
                  ((r.upperBoundType = 'inclusive' AND
                  r.upperBoundValue < 30) OR
                  (r.upperBoundType = 'exclusive' AND
                  r.upperBoundValue ≤ 30))))
```

(7.27)

URL	connectionId	flags
con.info	874	17
hw.ac.uk	98	25

Table 7.9: The result of executing Query (7.27) to the registry database instance in Table 7.5.

the registry database do not have an overlapping condition if their lower bound value is greater or equal to 30.

The second sub-query performs the measurement entailment test. This ensures that all of the continuous queries returned have the same or more restrictive condition on the measurement attribute latency. Again, care is needed with the data type (i.e. whether it is an integer or a real), the range boundary type (i.e. whether it is infinite, inclusive, or exclusive), and the value.

The result of posing Query (7.27) to the registry database instance given in Table 7.5 is given in Table 7.9. The results of this query are the continuous queries for which the republisher is relevant. However, the result should be the continuous queries for which the republisher is strongly relevant. This requires the registry service to perform a series of subsumption tests. For the example considered, the second result would not pass the strong relevance test since the query is part of a republisher and the query has a more restrictive condition as it restricts the values that the from attribute can take.

Once the continuous query agents have been identified by the registry service, they are each contacted and informed of the change in the publisher configuration. Upon receiving a notification of a change in the publisher configuration, the continuous query agent updates its list of relevant publishers. It then checks whether the publisher is maximal relevant for its query and if it is, the agent applies the results of Chapter 6 to test whether they need to do the following:

1. Amend their meta query plans.

2. If they have altered their meta query plan, check whether their current query plan is consistent with the new meta query plan.

Since continuous query agents are initially informed of all relevant publishers, and informed of any changes in the set of relevant publishers, they can maintain their meta query plans in all the cases presented in Chapter 6 without further interaction with the registry service. To patch their plan as required by Proposition 6.4 case 3, the continuous query agent need only maintain a list of relevant publishers.

7.4 Switching between Query Plans

The theory presented for query planning and plan maintenance has only considered how to create and update a meta query plan and query plan. For a production system, when a query plan is updated there needs to be a set of protocols to switch from the old query plan to the new plan in a controlled and well defined manner. These protocols should ensure, as far as possible, that the resulting answer streams are sound and complete with respect to their query, and are duplicate free and weakly ordered. Where it is not possible to ensure these properties, the system should be able to detect the situation and add a warning message to the result set to inform the user of the shortcoming.

A simple approach to switch between query plans is to perform the following actions:

- Temporarily stop all the streams affected.
- Move any state information from the existing node to the new node.
- Ensure all connections are updated.
- Start streaming again.

This approach has been implemented as a way of spreading the load across co-located machines in the Aurora system [76]. However, this approach is not suitable in general for a distributed stream system such as R-GMA. This is because there would be a substantial delay introduced in stopping all the streams, resulting in the system being unable to respond to user queries.

The following is an alternative approach that ensures the four properties of an answer stream are guaranteed or that the system detects when this is not the case.

The proposed mechanism has not been implemented since the purpose of the implementation was to show the correctness of the query planning and maintenance mechanisms.

There are five cases when a query plan needs to be updated if there is a change to the meta query plan. These are:

1. A producer is added to the meta query plan.
2. A producer is removed from the meta query plan.
3. A republisher is removed from the meta query plan which appears in an equivalence class with other republishers and is in the query plan.
4. A republisher is added to the meta query plan which has created a new equivalence class.
5. A republisher is removed from the meta query plan which was in an equivalence class on its own.

The situations in cases 1 and 2 have already been implemented in R-GMA. For case 1, each publisher caches a published tuple for a duration defined by the publisher's *retention period*. This provides the registry service with the time needed to contact the continuous query agents for which the producer is relevant and for them to then contact the producer agent and start streaming. For case 2, the producer is simply removed from the query plan.

The situation in case 3 is the one presented in the example of Section 6.1.4 when republisher R_1 is removed from publisher configuration $\mathcal{P}_2$. The meta query plan for q_1 in publisher configuration $\mathcal{P}_2$ was

$$\mathcal{M}_{q_1}(\mathcal{P}_2) = \Big(\big\{ \{ R_1, R_4 \} \big\}, \emptyset \Big). \tag{7.28}$$

It is assumed that the query plan being used was to contact republisher R_1. The result of performing the plan maintenance means that the meta query plan for q_1 in the new publisher configuration $\mathcal{P}_3$ is

$$\mathcal{M}_{q_1}(\mathcal{P}_3) = \Big(\big\{ \{ R_4 \} \big\}, \emptyset \Big). \tag{7.29}$$

Since q_1 had been using R_1 in its query plan, then it must switch to using republisher R_4.

The proposed mechanism requires that a continuous query agent maintains a *latest-state* buffer where it stores the most recent tuple received on each channel. When switching from R_1 to R_4, the consumer searches in its latest-state buffer for the oldest tuple t_{old} received from R_1. The consumer then requests that R_4 starts streaming from t^{τ}_{old}, the timestamp of t_{old}. The timestamp of t_{old} is used rather than the tuple itself as the streams are only weakly ordered, i.e. the tuples at R_4 could appear in a different order.

Upon receiving this message, R_4 consults the tuples in its publishing buffer and, providing that t^{τ}_{old} is still within its retention period, starts streaming all tuples with a timestamp equal to, or newer than t^{τ}_{old}. Otherwise it will start streaming from the oldest tuple in its buffer. On receiving the stream from R_4, the consumer must filter, on a per channel basis, the first part of the stream against its latest-state buffer. Only once it starts receiving tuples newer than the ones in its latest-state buffer does its answer stream start getting new tuples.

This mechanism ensures that the answers received by the consumers are sound with respect to the query. Providing that the tuples are still within the retention periods of the publishers involved, the answer stream will be complete. In the cases where the stream is not complete, a bound in time can be provided on the incompleteness in the answer stream, i.e. the time between t^{τ}_{old} and the retention period of the republisher. Due to the filtering based on the latest-state buffer the answer stream will be duplicate free, and weak order is guaranteed by the construction of the query plans.

The situation in case 3 is a special case of cases 4 and 5, where all of the channels which are changing their source publisher are switching to the same new source publisher. In cases 4 and 5 the situation is a lot more complex. However, for every channel that is changing its data source the timestamp of the most recently seen tuple on that channel can be sent to the new publisher. Obviously, there are some implementation issues that need to be addressed. For example, the continuous query agent would need to identify which publisher will be supplying which channel. This is not necessarily a trivial task.

7.5 Summary

This chapter has presented details of a prototype implementation of the proposed stream integration system, and the query planning and maintenance techniques developed in this thesis. These mechanisms have been implemented as an extension to the R-GMA Grid information and monitoring system as this provided much of the infrastructure required by a stream integration system. The implementation highlighted the fact that protocols will be required to manage the transition from one query plan to another when there is a change in the publisher configuration. The next chapter will conduct experiments to investigate aspects of the performance of the implementation.

Chapter 8

Performance Measures

This chapter investigates some key aspects of the performance of the proposed stream integration system and the associated query planning and maintenance techniques, specifically:

1. The performance of the registry service when identifying the relevant components for a new registration, e.g. the relevant publishers for a new continuous query.

2. The effects of a hierarchy of publishers on the latency of an answer tuple.

These are important issues that affect the overall performance of the stream integration system and thus would affect the take-up of such a system.

The investigation into the performance of the registry service will be presented in Section 8.1. The registry service is a key component in the stream integration system that handles the registration of all the new producers, consumers, and republishers. Part of this process involves identifying those registered services that are relevant for the new producer, consumer, or republisher. The performance of the registry service is of great importance as delays would affect the entire stream integration system. The experiments investigate the two approaches, detailed in Chapter 7, to the task of identifying relevant consumers for the registration of a new publisher.

Section 8.2 investigates the effects of introducing a hierarchy of publishers on the latency of an answer tuple. It is expected that the introduction of the hierarchy will increase the time taken for a tuple to be delivered to a consumer since it must pass through some number of republishers en route. However, there are benefits to using

a hierarchy of publishers, e.g. a continuous query need not contact every producer of relevant information. The second set of experiments aims to quantify the effect of the hierarchy of publishers on the latency of an answer tuple and will investigate the effect of introducing more than one level of republishers on the time taken.

All of the tests were carried out on up to six identical machines running Fedora Core 2. Each machine had a 1.4 GHz Intel Pentium 4 processor with 256 KB of Cache, 256 MB of RAM, and connected by a 100 Mbps LAN. Tomcat version 5.0.28 was used together with Sun's Java 1.4.2 version 8.

8.1 Performance of the Registry Service

The registry service is a key component of the stream integration system proposed in Chapter 4 and as such its performance will have a large impact on the system as a whole. Four closely related tasks that the registry service performs on a regular basis are:

1. Identifying relevant consumers and republishers when a new publisher registers.
2. Identifying relevant consumers and republishers when an existing publisher is removed from the system.
3. Identifying relevant publishers when a new consumer or republisher registers.
4. Identifying relevant publishers when an existing consumer or republisher is removed from the system.

Each of these tasks involves performing tests on the registered producers, consumers, and republishers, as appropriate. For the Grid information and monitoring application this will involve a large number of tests. Thus, the registry service needs an efficient method to perform these relevance tests.

As discussed in Section 7.2, the current R-GMA system has two approaches to the relevance test depending on whether it is identifying publishers or continuous queries. The first approach, used when identifying relevant producers for a query, is to generate a query that retrieves the relevant producers from the registry service database, i.e. the query to the registry database performs the relevance test. The second approach,

used when identifying relevant continuous queries, is to retrieve all continuous queries that involve the same global relation from the registry service database and perform the relevance test in the registry service code. However, when republishers can be used to answer a continuous query, the techniques to handle continuous queries and publishers needs to be consistent. Thus, performance measures were conducted to investigate which of the approaches is more efficient and should be adopted for the implementation of the query planning mechanisms developed in this thesis.

It is worth noting that the experiments were performed before republishers were permitted to answer a continuous query. Thus, the relevance test only checked the satisfiability of the view and query conditions.

8.1.1 Experimental Method

The two approaches to performing the relevance test were investigated for the case of identifying relevant continuous queries for a new producer. The tests would measure the time taken by the two approaches to identify all the relevant continuous queries.

The first approach considered involves retrieving all of the consumers from the registry service database that have a query which involves the relation in the view of the producer and then performing the satisfiability test in the registry service. This is the method implemented in the current R-GMA system, so the existing registry service could be used for the experiments.

The R-GMA registry service has an efficient implementation of the satisfiability test, the test does not necessarily need to compare every pair of conjuncts in the view of the producer and the query of the consumer. As soon as one attribute is found to be unsatisfiable the consumer is discarded and the next one is considered. Similarly, if either the query of the consumer or the view of the producer does not have a condition then the test will return true immediately as any query condition considered is satisfiable with the true condition.

The experiments conducted measure the time taken by the R-GMA registry service to perform the following tasks:

1. Generate the query to the registry service database to retrieve all the consumers for the global relation named in the view of the producer.

2. Pose the generated query to database.

3. Parse the condition of each consumer returned by the registry database query and perform the satisfiability test.

The second approach considered involves performing the satisfiability test as part of the query posed to the registry service database. This required the development of an extension to the registry service, referred to as R-GMA′, which used a database that stores the conditions in the query of the consumer in a structured format. Full details of the database schema used can be found in Section 7.3.1. The R-GMA′ registry service would then generate a suitable query to perform the satisfiability test when a new producer registered.

The experiments conducted measure the time taken by the R-GMA′ registry service to perform the following tasks:

1. Generate the query to the registry service database that would retrieve the consumers that are relevant to the producer, i.e. the query would perform the satisfiability test.

2. Pose the generated query to the database.

8.1.2 Experimental Setup

The experiments to measure the performance of the two registry services were run on four machines. The machines were configured so that one machine acted as the schema service, one machine as the registry service, one machine as a consumer service which also hosted all the consumers, and one machine as a producer service which also hosted the producers.

The tests were conducted by annotating the code of each registry service so that it would write a timestamped log message before entering the method to identify the relevant consumers and a similar message once the method had completed. The time taken to perform the relevance test was the difference between the timestamps.

To conduct the experiments, the global schema was extended to include the new

relation

$$\mathsf{regTest}(\underline{\mathsf{hostname}} :: \mathsf{string},\ \mathsf{testColumn1} :: \mathsf{string},\ \mathsf{testColumn2} :: \mathsf{integer},\ \mathsf{testColumn3} :: \mathsf{real},\ \mathsf{testColumn4} :: \mathsf{string}, \ldots, \mathsf{testColumni}, \ldots, \mathsf{testColumn19} :: \mathsf{string}), \quad (8.1)$$

where the type of each attribute is given and testColumn$i$ for $i \in 4..19$ had the type string. This relation allowed query and view conditions to involve many attributes.

Three sets of experiments were conducted to investigate the time taken to find the relevant consumers for a new producer under different experimental conditions. The different conditions considered were:

1. All of the consumers in the system are relevant for the new producer.

2. None of the consumers in the system are relevant for the new producer.

3. A mixture of relevant and non-relevant consumers for the new producer exist in the system.

The first two cases provide extreme cases for the experiment and will provide upper and lower bounds for the performance of the registry services. The third case matches the normal situation where some of the consumers will be relevant for a new producer and others will not.

For each run of the experiment, 50 consumers were instantiated. Once all of the consumers had registered with the registry service, 10 producers were added to the system one at a time with the average of the time taken to add a producer recorded. This was repeated multiple times for each of the different experimental conditions varying the number of conditions in the producer view and the consumer query respectively, each varying from 0 conditions to 10 conditions. Varying the number of conditions alters the amount of work that has to be conducted to perform the satisfiability test.

Another way of increasing the amount of work performed by the satisfiability test is to increase the number of consumers. Thus, the experiment was repeated with 100 consumers, all of which were relevant for the new producers added.

In all of the experiments, the producers registered the same view. The view was constructed from the following 10 conditions:

$$\begin{aligned} &\mathsf{testColumn1} = \texttt{'constant'} \ \wedge \ \mathsf{testColumn2} = 25 \ \wedge \ \mathsf{testColumn3} = 5.0 \ \wedge \\ &\mathsf{testColumn4} = \texttt{'xxx'} \ \wedge \ \mathsf{testColumn5} = \texttt{'yyy'} \ \wedge \ \mathsf{testColumn6} = \texttt{'zzz'} \ \wedge \\ &\mathsf{testColumn7} = \texttt{'aaa'} \ \wedge \ \mathsf{testColumn8} = \texttt{'bbb'} \ \wedge \\ &\mathsf{testColumn9} = \texttt{'ccc'} \ \wedge \ \mathsf{testColumn10} = \texttt{'ddd'}. \end{aligned} \tag{8.2}$$

Each of the conditions was added in turn, so when the producers registered a view with four conditions they registered the view:

$$\begin{aligned} &\mathsf{testColumn1} = \texttt{'constant'} \ \wedge \ \mathsf{testColumn2} = 25 \ \wedge \\ &\mathsf{testColumn3} = 5.0 \ \wedge \ \mathsf{testColumn4} = \texttt{'xxx'}. \end{aligned} \tag{8.3}$$

The queries of the consumers for each of the experiments will be given in the relevant results section below.

8.1.3 Results

The following sections present the results of running the experiments for each of the experimental conditions in turn.

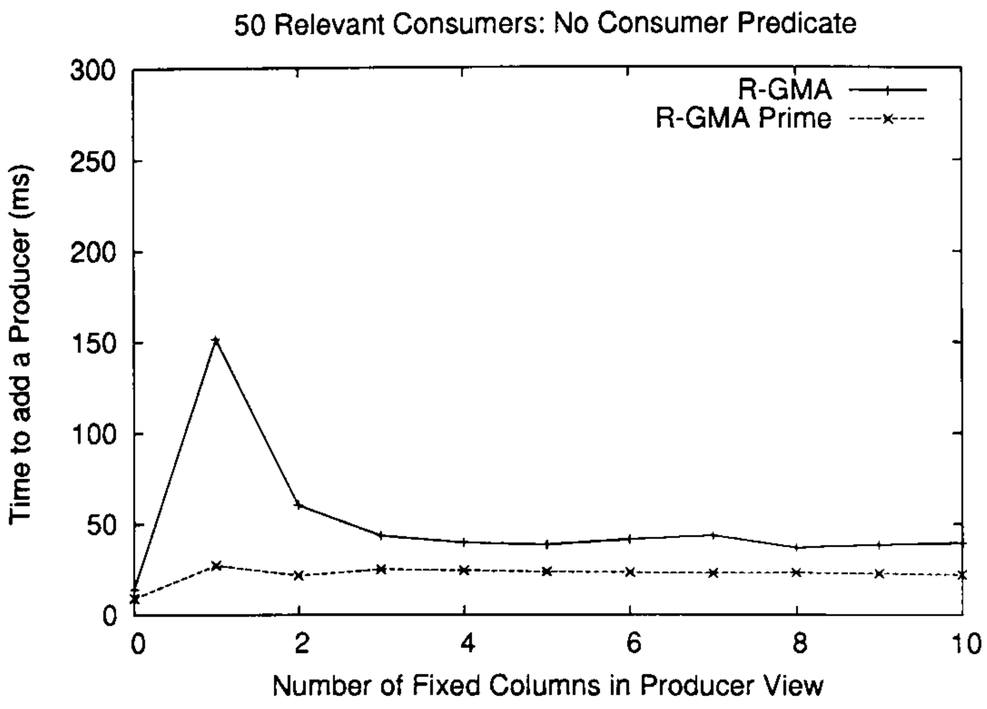
Experiment with 50 Relevant Consumers

The 10 query conditions for the relevant consumers were:

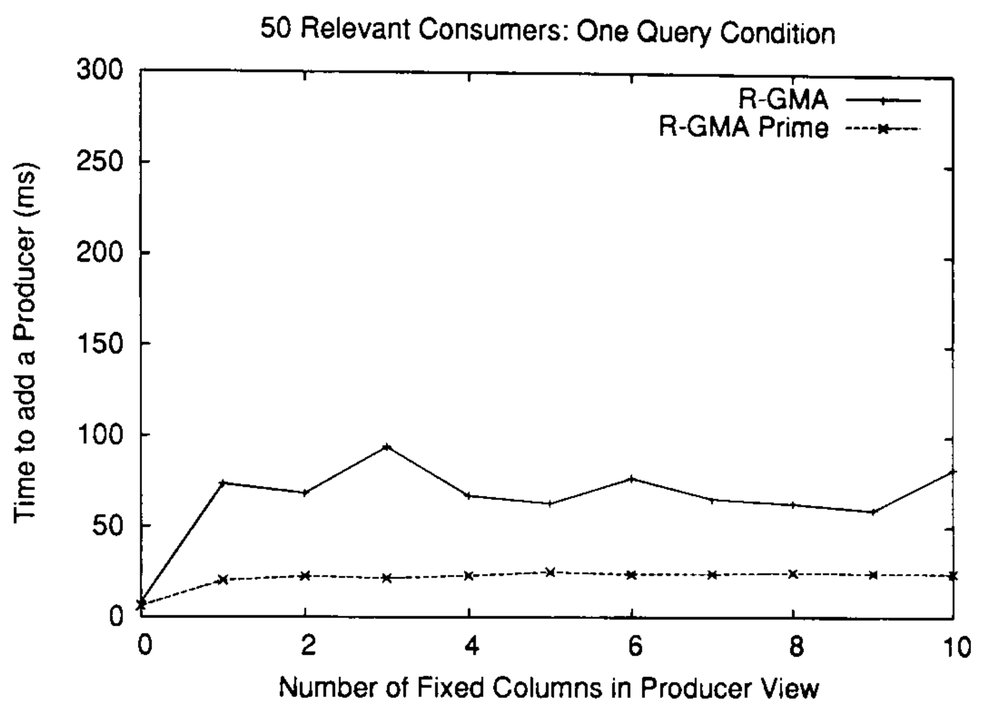
$$\begin{aligned} &\mathsf{testColumn1} = \texttt{'constant'} \ \wedge \ \mathsf{testColumn2} \leq 50 \ \wedge \ \mathsf{testColumn3} < 10.5 \ \wedge \\ &\mathsf{testColumn2} > 0 \ \wedge \ \mathsf{testColumn4} = \texttt{'xxx'} \ \wedge \ \mathsf{testColumn5} = \texttt{'yyy'} \ \wedge \\ &\mathsf{testColumn3} \geq 2.5 \ \wedge \ \mathsf{testColumn6} = \texttt{'zzz'} \ \wedge \\ &\mathsf{testColumn7} = \texttt{'aaa'} \ \wedge \ \mathsf{testColumn8} = \texttt{'bbb'}. \end{aligned} \tag{8.4}$$

Again, the query with 1 condition consists of the first condition, the query with 2 conditions the first two, and so on.

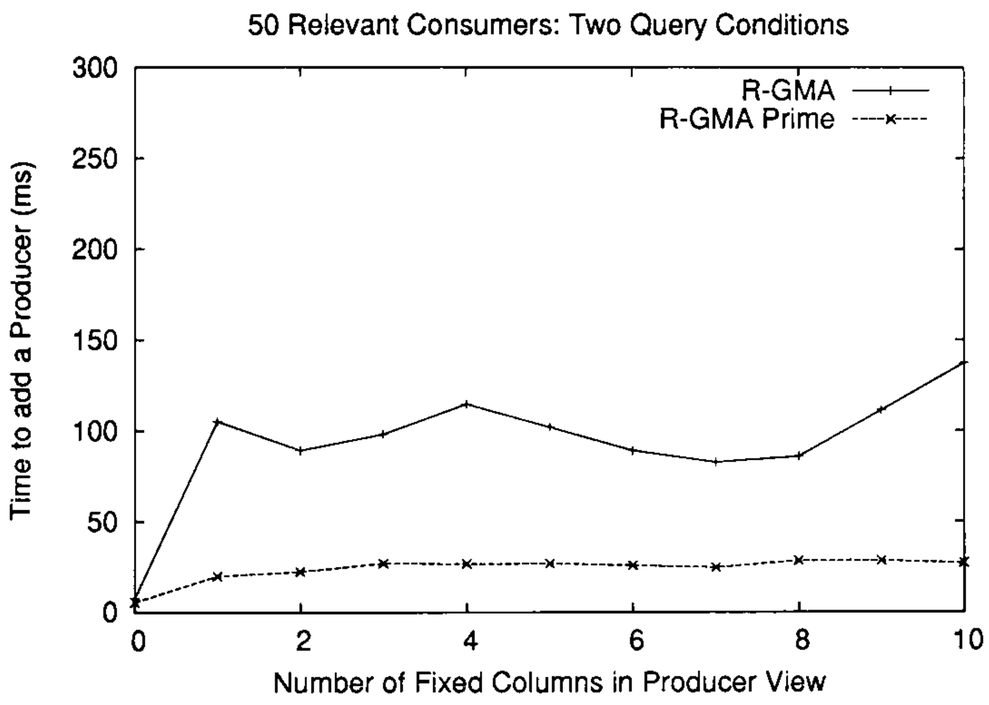
Figures 8.1 and 8.2 present the results for the experiments when there are 50 relevant consumers for the new producers being added. The results show that the R-GMA′ registry service significantly outperforms the R-GMA registry service.



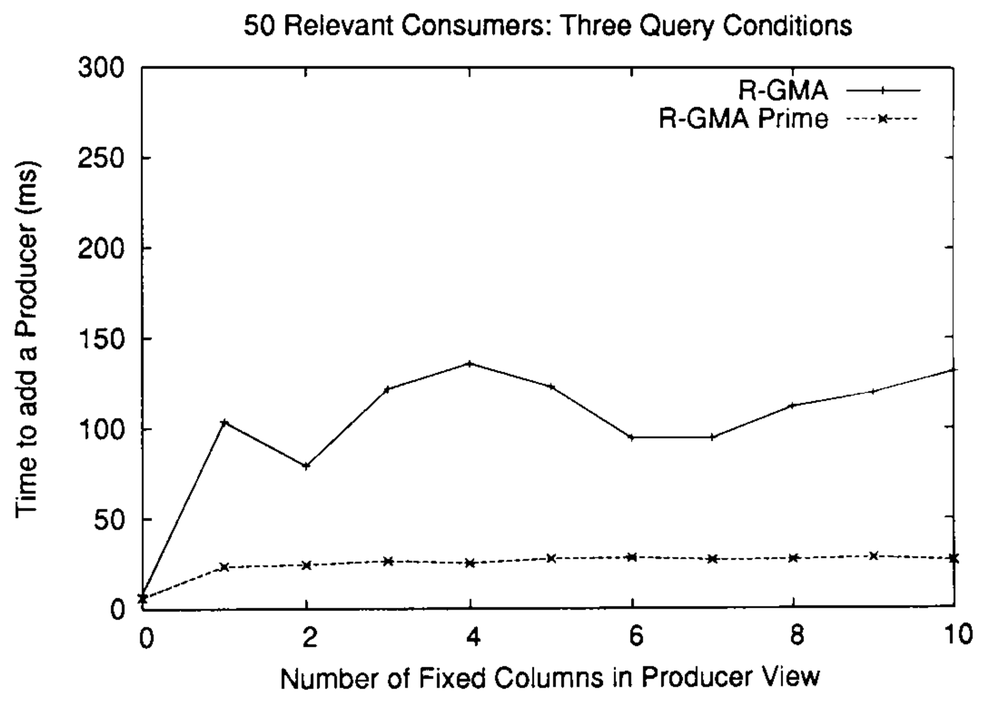
(a) No consumer predicate.



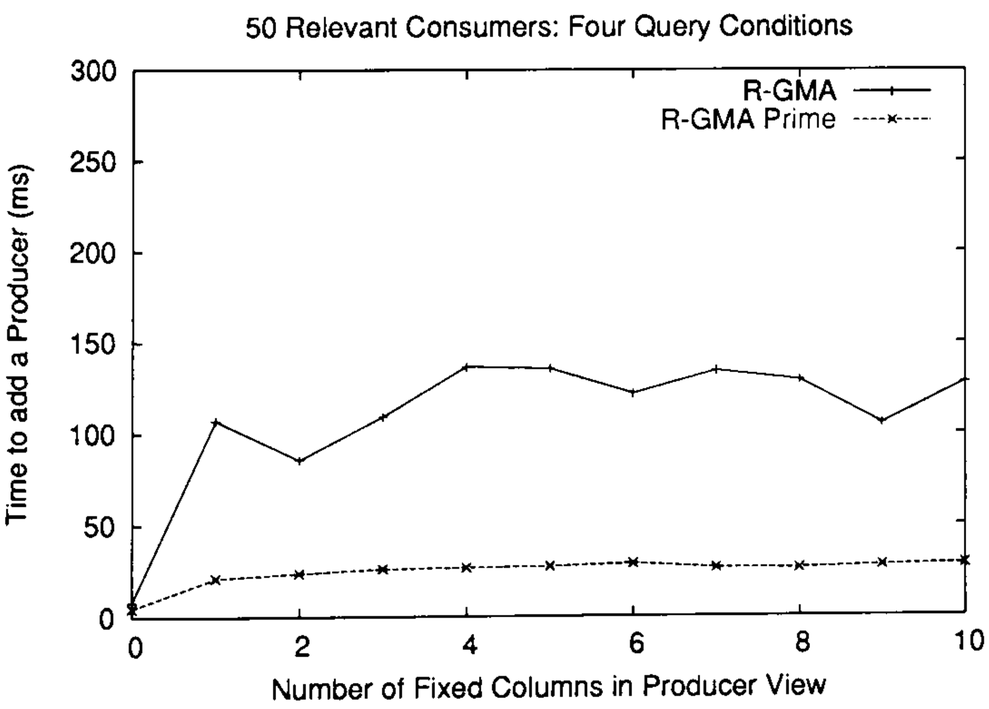
(b) One consumer predicate.



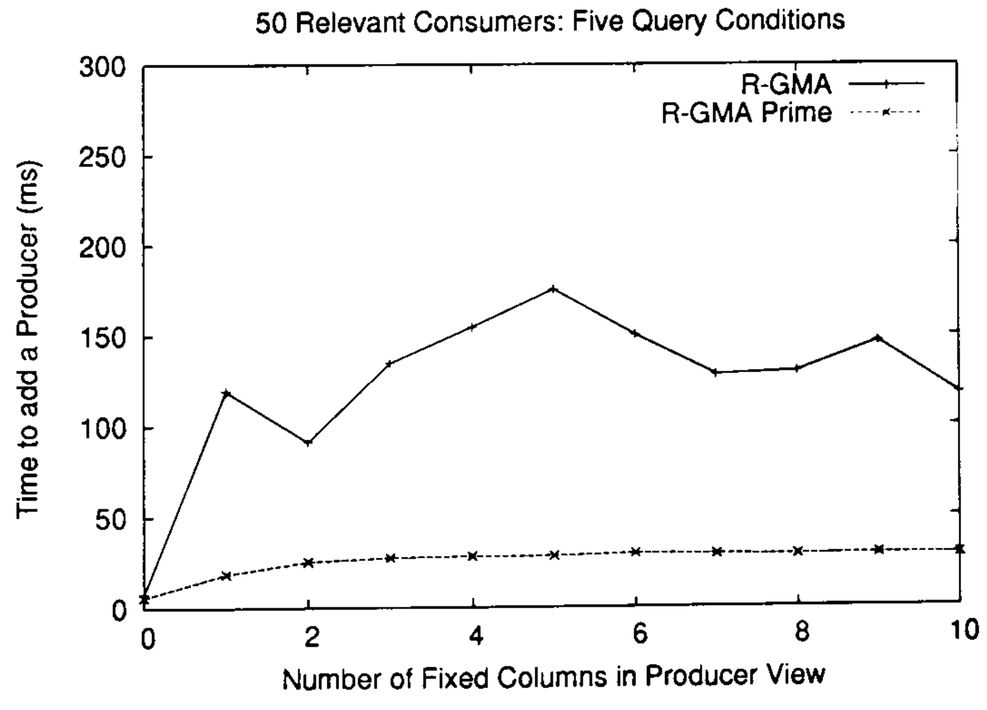
(c) Two consumer predicates.



(d) Three consumer predicates.

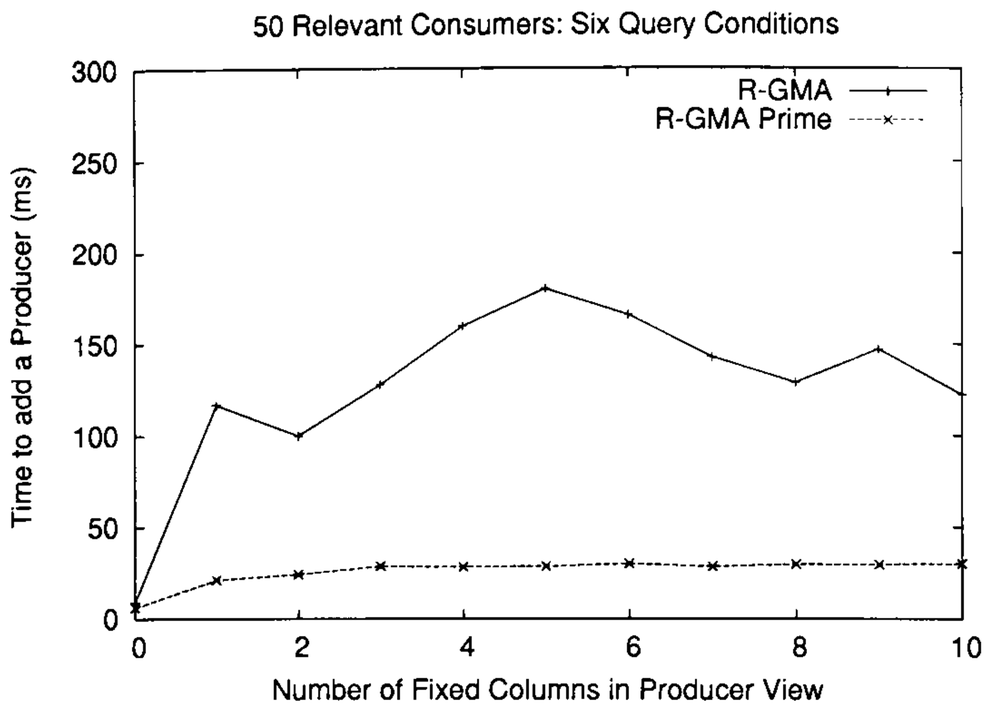


(e) Four consumer predicates.

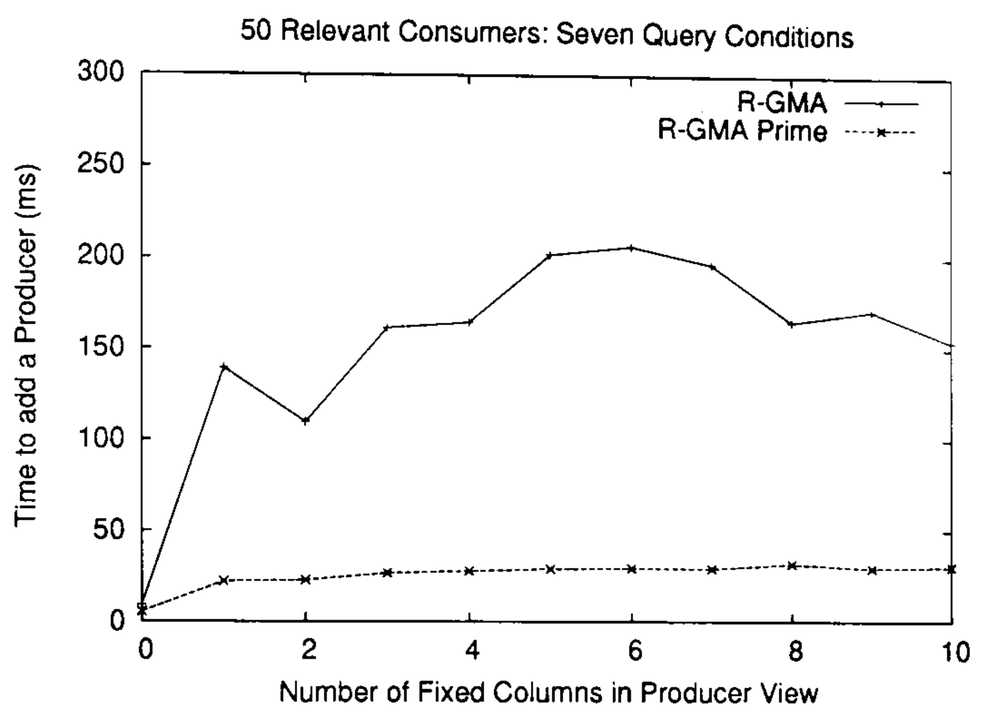


(f) Five consumer predicates.

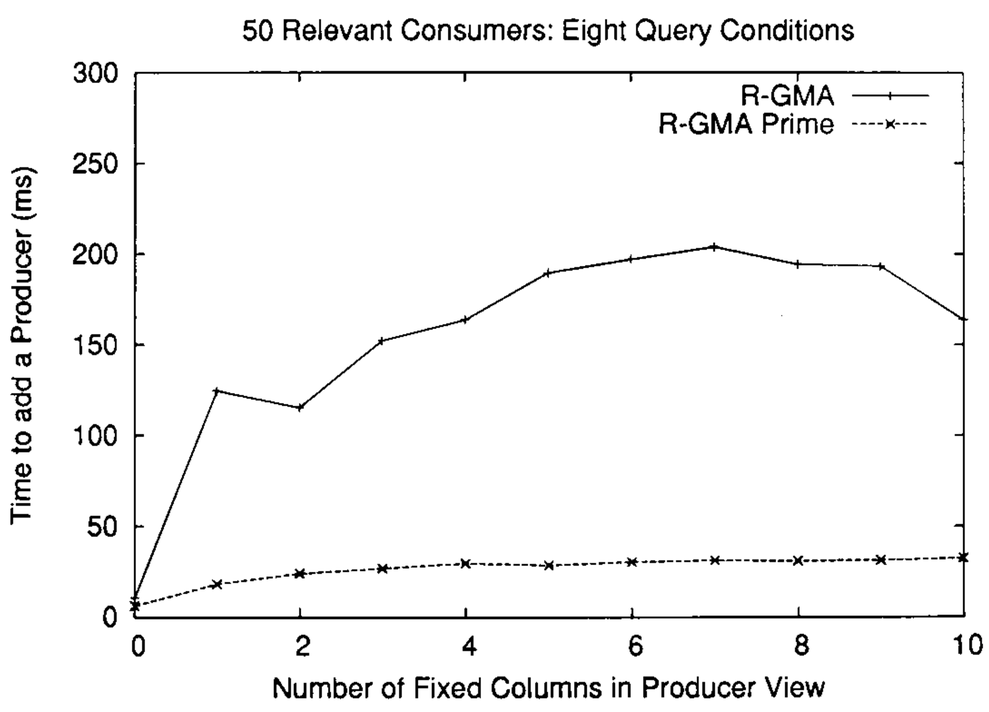
Figure 8.1: Time taken to identify relevant consumers for a new producer when there are 50 registered consumers, all of which are relevant.



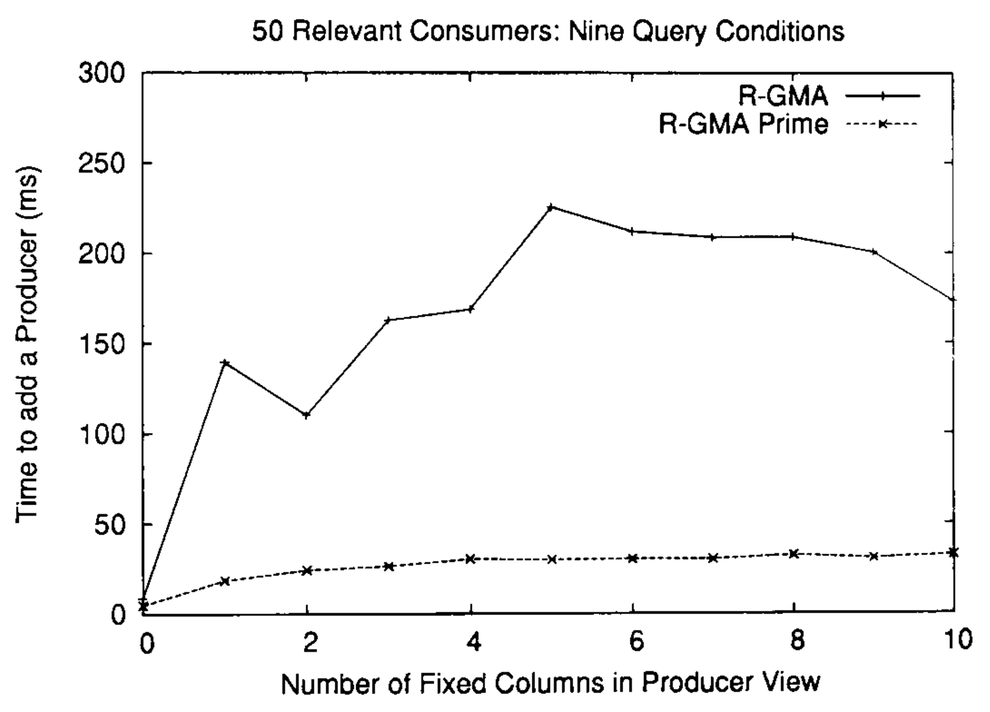
(a) Six consumer predicates.



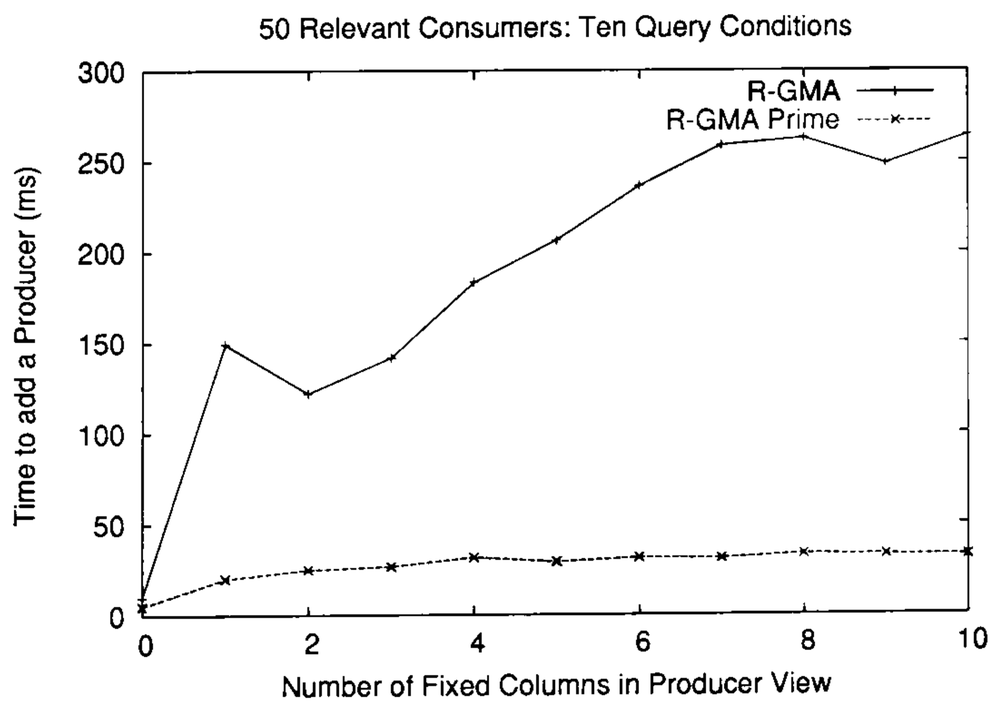
(b) Seven consumer predicates.



(c) Eight consumer predicates.



(d) Nine consumer predicates.



(e) Ten consumer predicates.

Figure 8.2: Time taken to identify relevant consumers for a new producer when there are 50 registered consumers, all of which are relevant.

Number of query conditions	R-GMA		R-GMA′	
	Average	Variance	Average	Variance
0	49.8	35.4	22.2	4.6
1	65.5	21.6	21.7	5.4
2	92.9	32.4	23.7	6.6
3	102.0	35.7	24.4	6.3
4	109.0	37.1	24.2	6.9
5	122.9	44.3	25.2	7.2
6	127.4	45.7	25.9	7.1
7	152.6	55.3	25.8	7.4
8	155.2	56.5	26.2	7.9
9	165.2	62.6	26.2	8.2
10	189.1	79.0	27.0	8.4

Table 8.1: The mean and variance for 50 relevant consumers.

In all of the graphs, the R-GMA′ registry service shows near constant performance of between 20 and 30 ms. It is more difficult to characterise the performance of the R-GMA registry service due to the variability in its performance. The average and variance of the plots are given in Table 8.1 and are also shown in Figure 8.12(a). The average values for the R-GMA′ registry service support the claim that it shows near constant performance, although there is a slight increase in the time taken as the number of conditions in the queries of the consumers increase. The average values for R-GMA registry service show that as the number of conditions increases, the time taken to identify the relevant consumers increases significantly.

The performance of the two registry services is most similar in the case where the consumers have no predicate, Figure 8.1(a). In this case, the R-GMA registry service can exploit an optimisation in the satisfiability test used to check relevance; when testing any condition against no predicate the result is always true. Thus, once the registry service has ascertained that the consumer's query does not contain a predicate it need not perform a satisfiability test. Even in this case, the R-GMA′ registry service is about twice as fast as that of the R-GMA registry service. This is because the R-GMA registry service must process each consumer to check it has no predicate whereas the

R-GMA′ registry service can simply return the result set from the database query.

A common feature of the R-GMA registry service performance, that occurs in all of the graphs, is that it takes longer to process a producer registering a view with one condition than it does with two. This is most prominently shown in Figure 8.1(a).

Experiment with 50 Non-relevant Consumers

The 10 query conditions for the non-relevant consumers were:

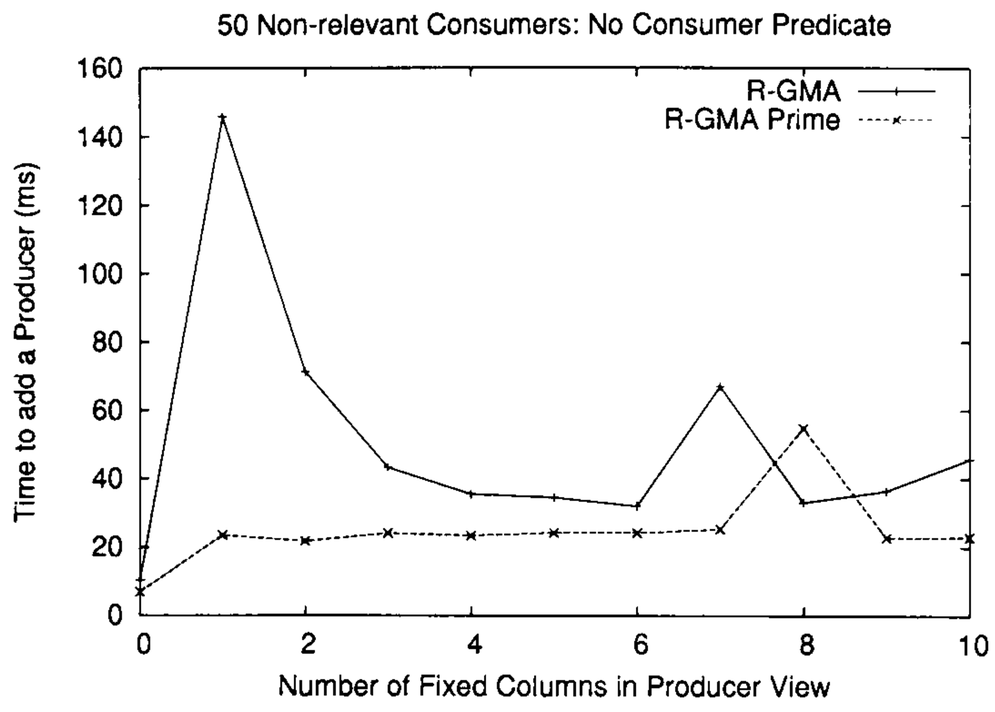
$$\begin{aligned} &\mathsf{testColumn1} = \texttt{'const'} \ \wedge \ \mathsf{testColumn2} > 50 \ \wedge \ \mathsf{testColumn3} \geq 10.5 \ \wedge \\ &\mathsf{testColumn2} \leq 100 \ \wedge \ \mathsf{testColumn4} = \texttt{'aaa'} \ \wedge \ \mathsf{testColumn5} = \texttt{'bbb'} \ \wedge \\ &\mathsf{testColumn3} < 25.5 \ \wedge \ \mathsf{testColumn6} = \texttt{'ccc'} \ \wedge \\ &\mathsf{testColumn7} = \texttt{'ddd'} \ \wedge \ \mathsf{testColumn8} = \texttt{'eee'}. \end{aligned} \tag{8.5}$$

Again, the query with 1 condition consists of the first condition, the query with 2 conditions the first two, and so on.

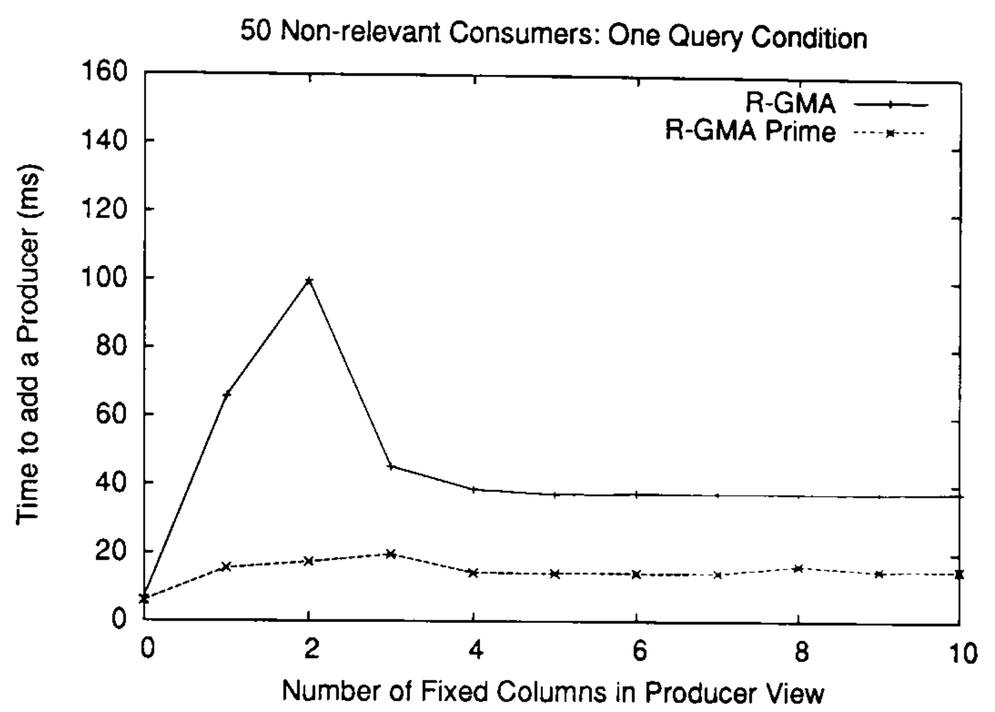
Figures 8.3 and 8.4 present the results for the experiments when there are 50 non-relevant consumers for the new producers being added. Again, the results show that the R-GMA′ registry service outperforms the R-GMA registry service. The following are thought to be anomalous readings:

- Figure 8.3(a) the R-GMA registry service with 7 conditions in the views of the producers.
- Figure 8.3(a) the R-GMA′ registry service with 8 conditions in the views of the producers.
- Figure 8.4(e) the R-GMA registry service with 7 conditions in the views of the producers.

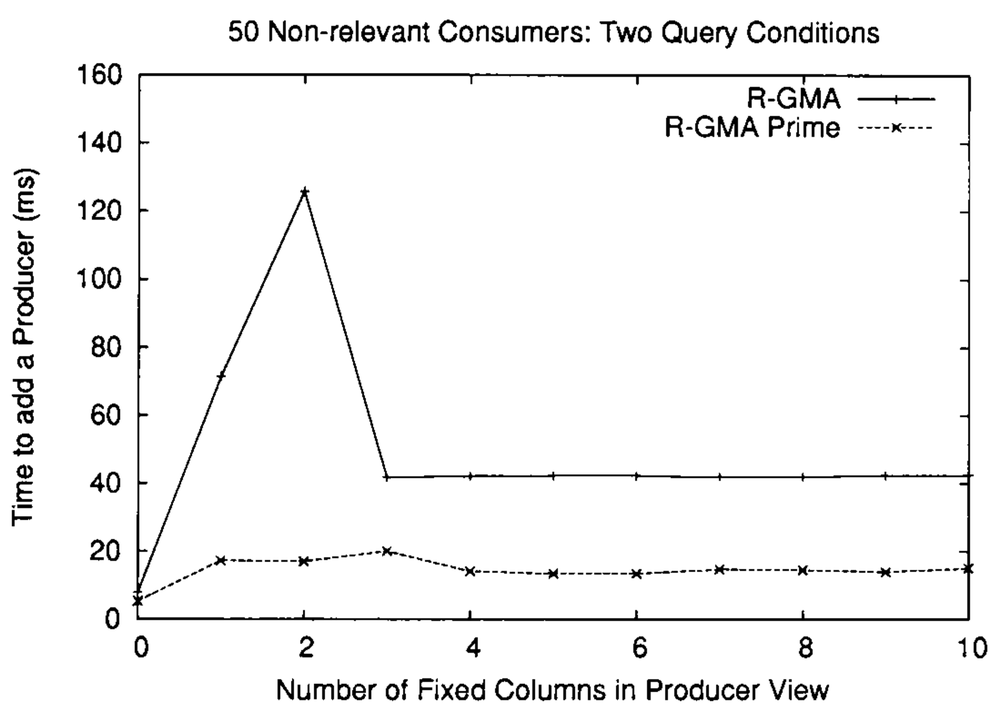
The average and variance of the plots are given in Table 8.2 and are also shown in Figure 8.7(a). For the R-GMA′ registry service, these show that after the case where neither the producer nor the consumer has any predicate, the performance is almost steady at 14 ms. For the R-GMA registry service, after the initial case, as the number of conditions increases the time taken to identify the relevant consumers increases. This increase is despite the optimisation in the satisfiability test whereby



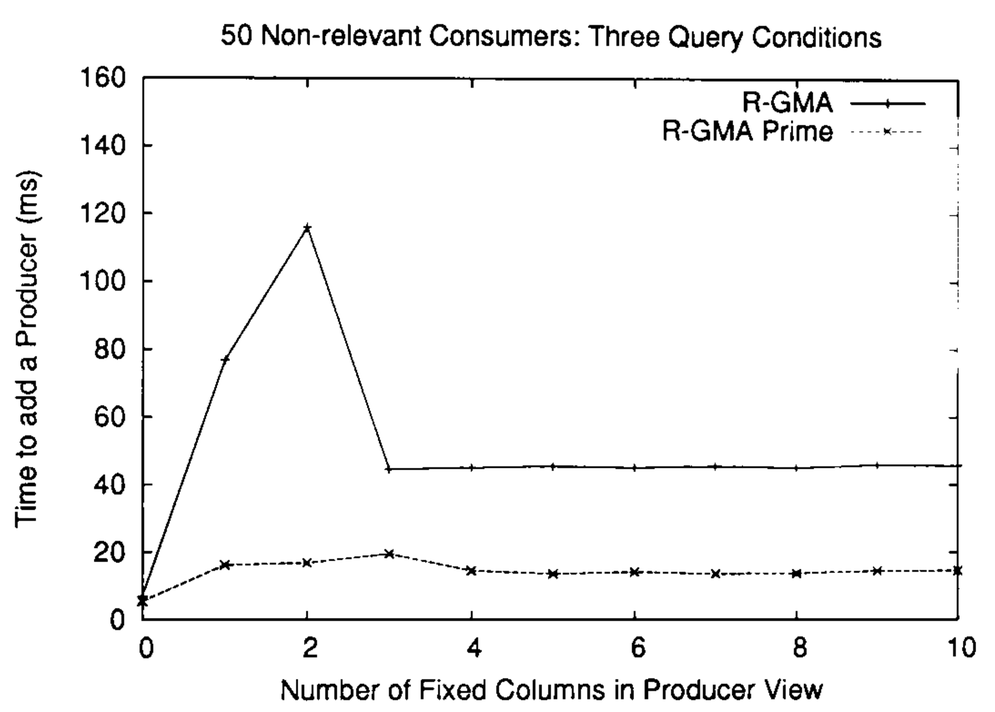
(a) No Consumer predicate.



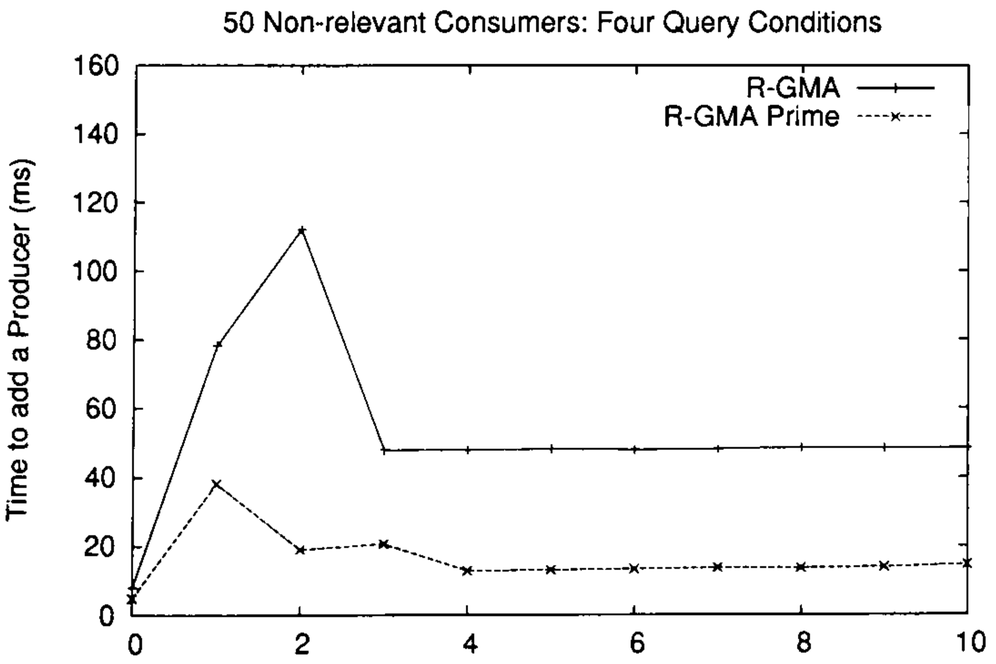
(b) One condition.



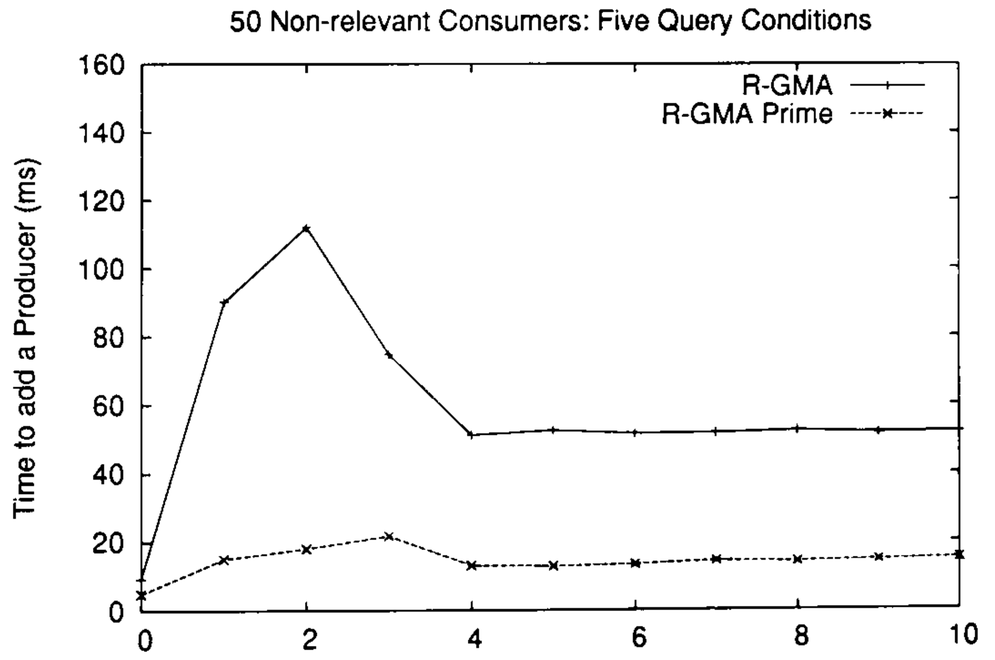
(c) Two conditions.



(d) Three conditions.

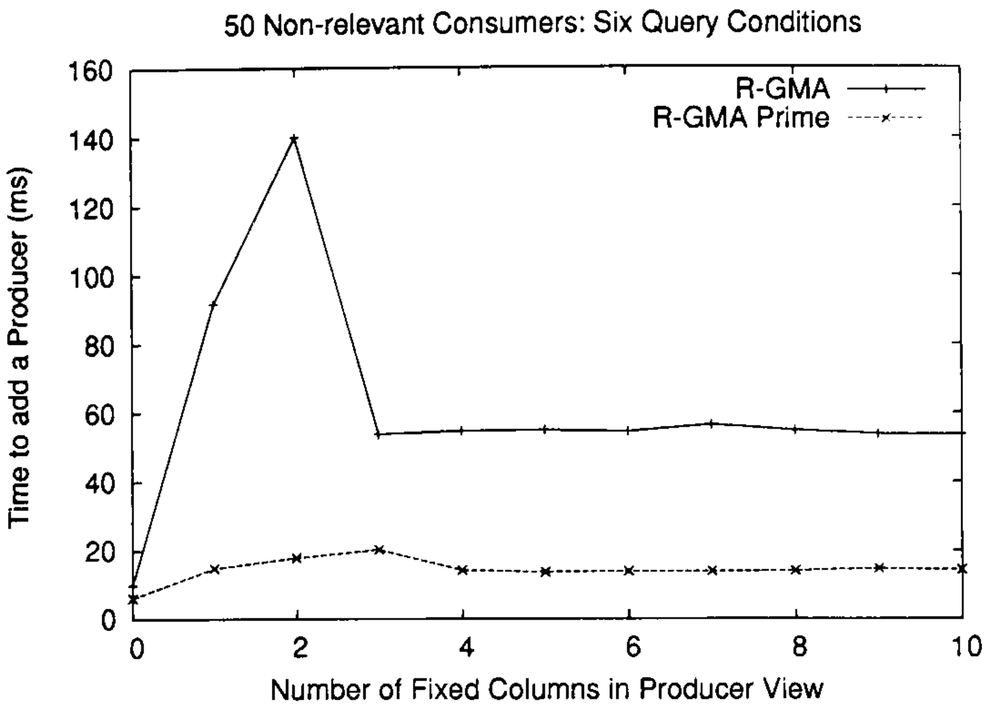


(e) Four conditions.

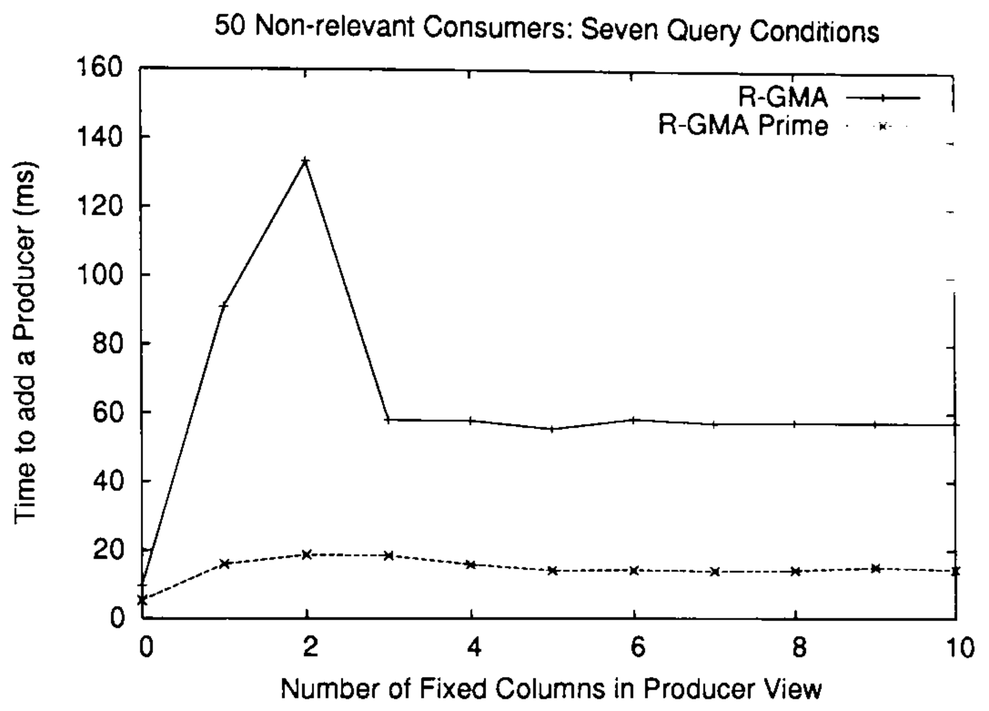


(f) Five conditions.

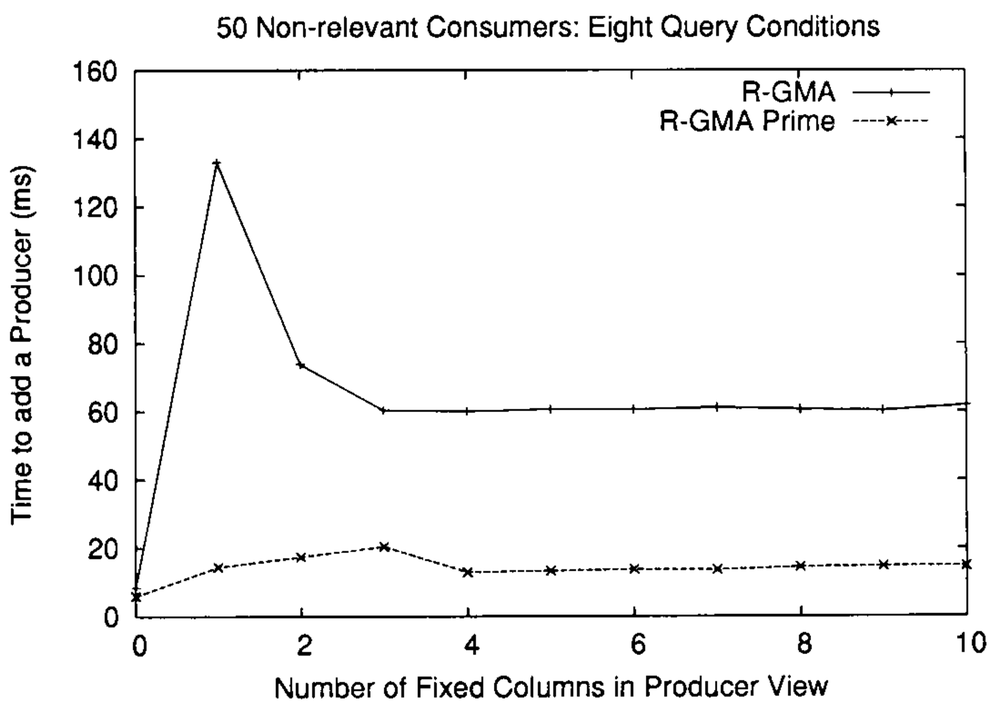
Figure 8.3: Results from registry service performance test with 50 non-relevant consumers.



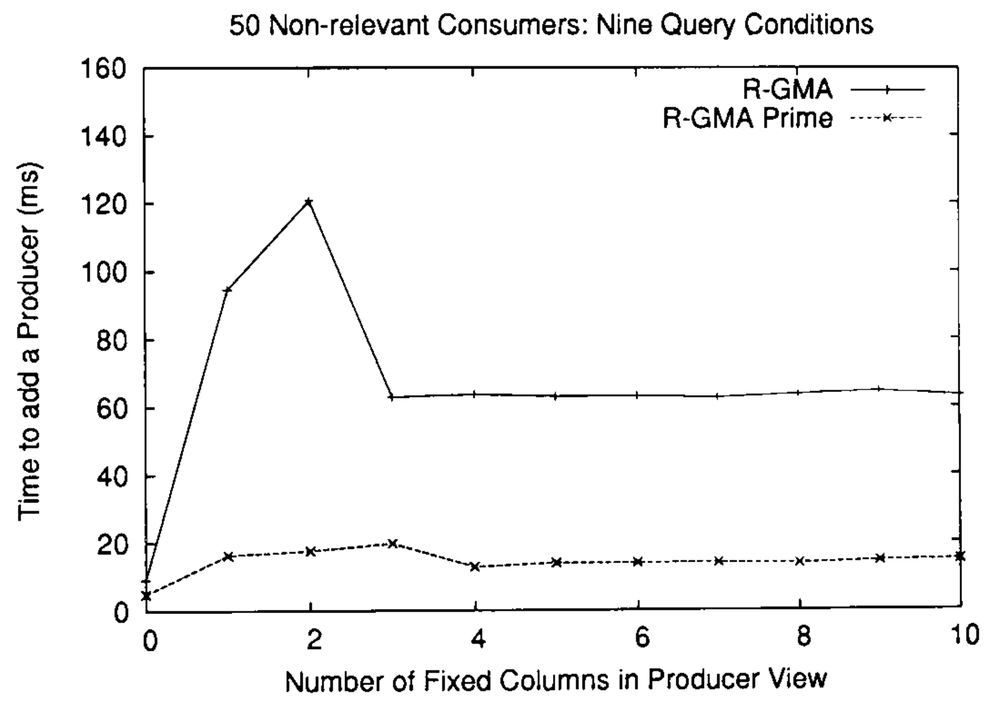
(a) Six conditions.



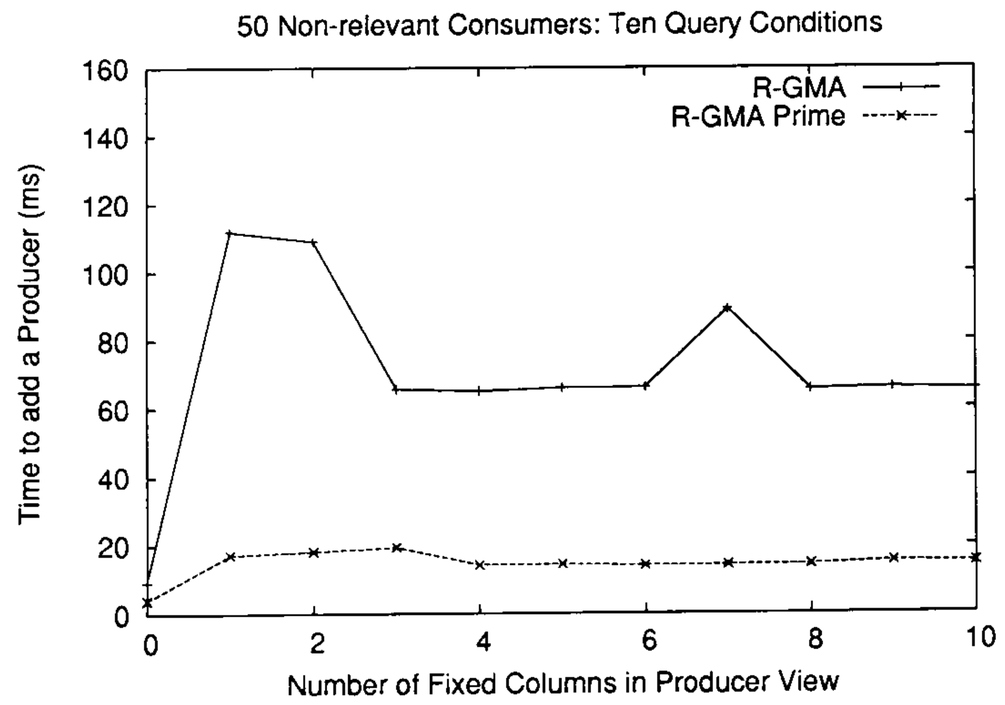
(b) Seven conditions.



(c) Eight conditions.



(d) Nine conditions.



(e) Ten conditions.

Figure 8.4: Results from registry service performance test with 50 non-relevant consumers.

Number of query conditions	R-GMA		R-GMA′	
	Average	Variance	Average	Variance
0	50.4	35.8	24.8	11.1
1	43.7	22.9	14.5	3.3
2	49.2	29.1	14.4	3.6
3	51.0	26.6	14.2	3.4
4	53.0	25.2	16.0	8.3
5	59.0	26.2	14.2	4.0
6	61.6	31.8	14.2	3.4
7	62.9	29.7	14.4	3.5
8	63.6	28.3	14.1	3.5
9	66.4	26.7	14.2	3.7
10	70.8	27.3	14.5	3.9

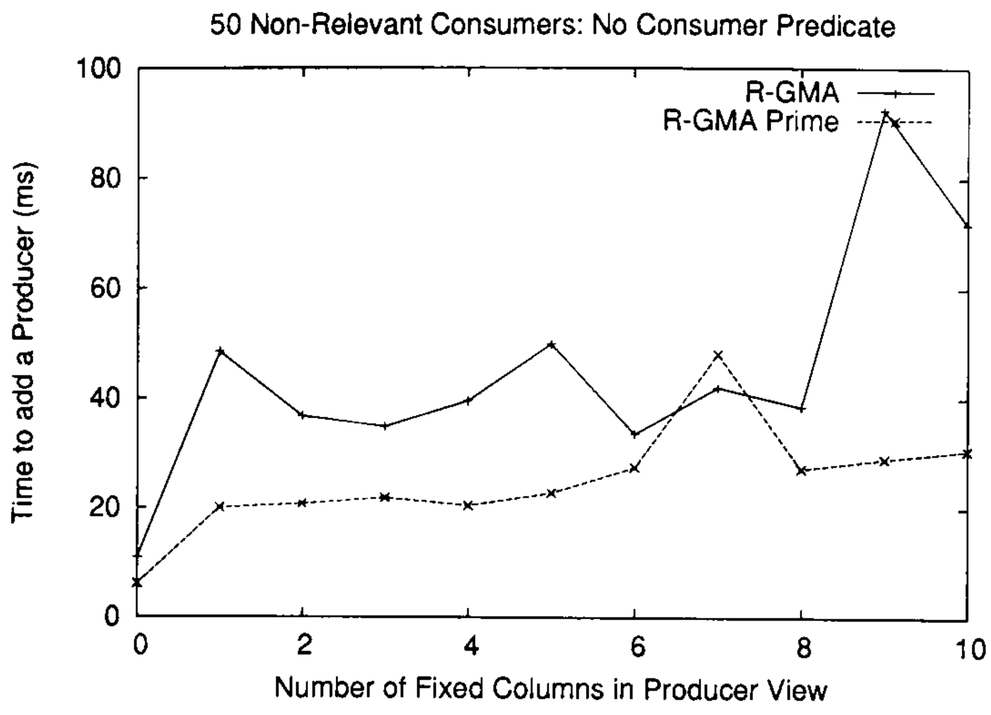
Table 8.2: The mean and variance for 50 non-relevant consumers.

when one condition is found to be contradictory the entire test fails. However, it can be explained since as the number of conditions in the queries of the consumers increases the longer it will take to parse the query expression before performing the satisfiability test.

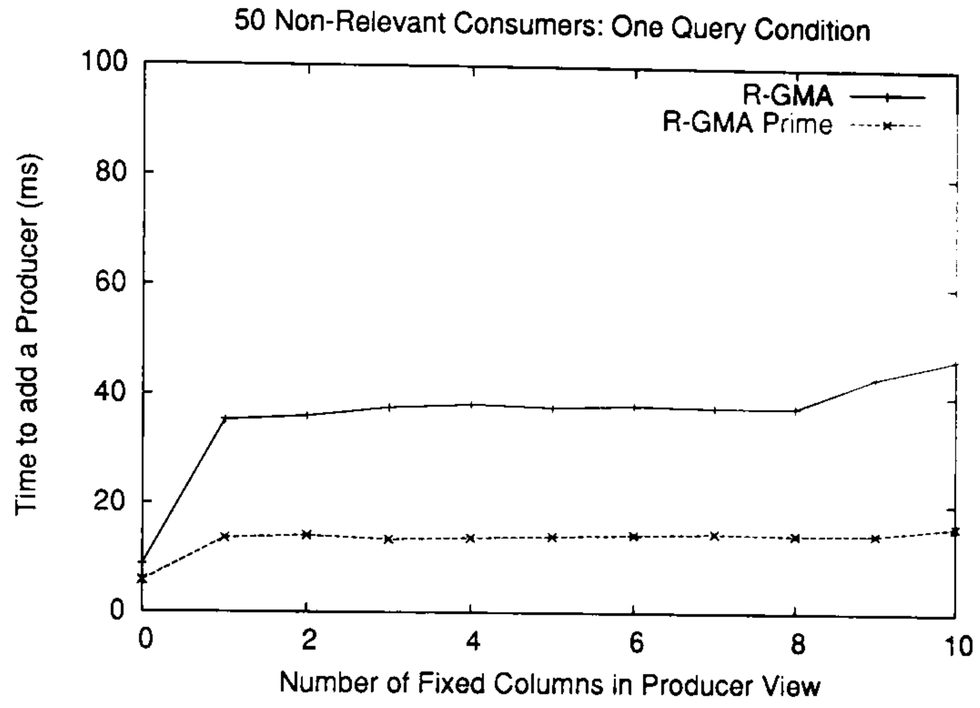
The characteristic feature of the graphs for the R-GMA registry service is the spike on the left hand side which peaks when the views of the producers have either one or two conditions. The graphs then slope down and when the views of the producers have four conditions have levelled off, although the value at which the graphs level off increases as the number of conditions in the queries of the consumers increases.

Since the spike at the start of the R-GMA registry service was more prominent under these experimental conditions it was decided to further investigate this feature. To understand whether the spike was caused by setting up the experiment or was a feature of the results, the experiment was run in reverse, i.e. first of all the producers with 10 conditions in their view were added, then 9, and so on until they had no conditions in their view. The results for these experiments are presented in Figures 8.5 and 8.6.

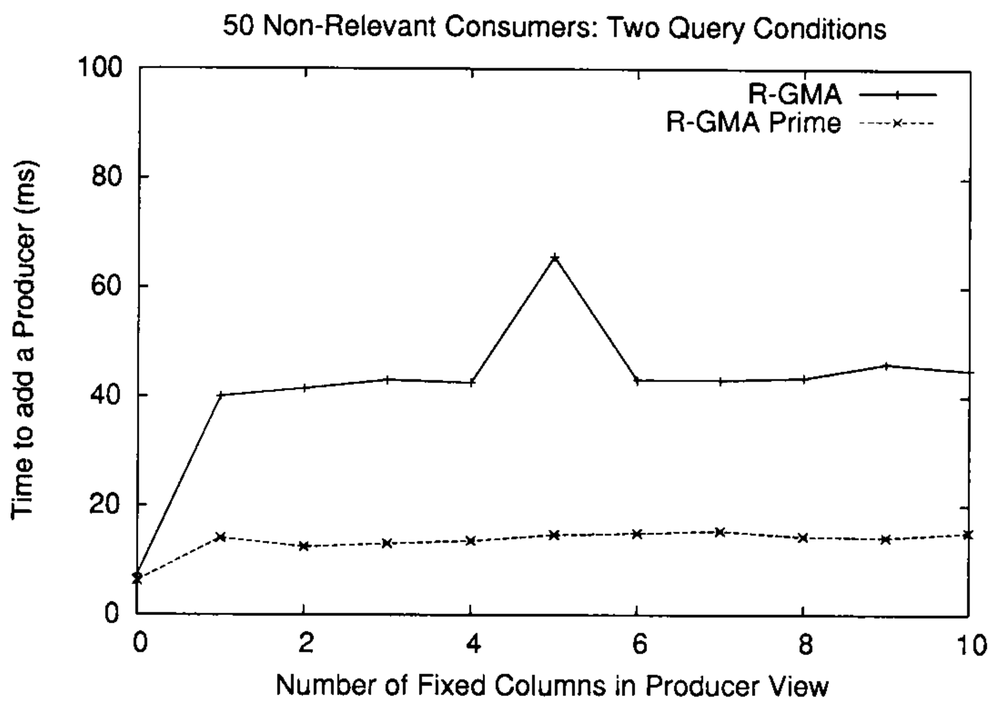
The following are believed anomalous readings:



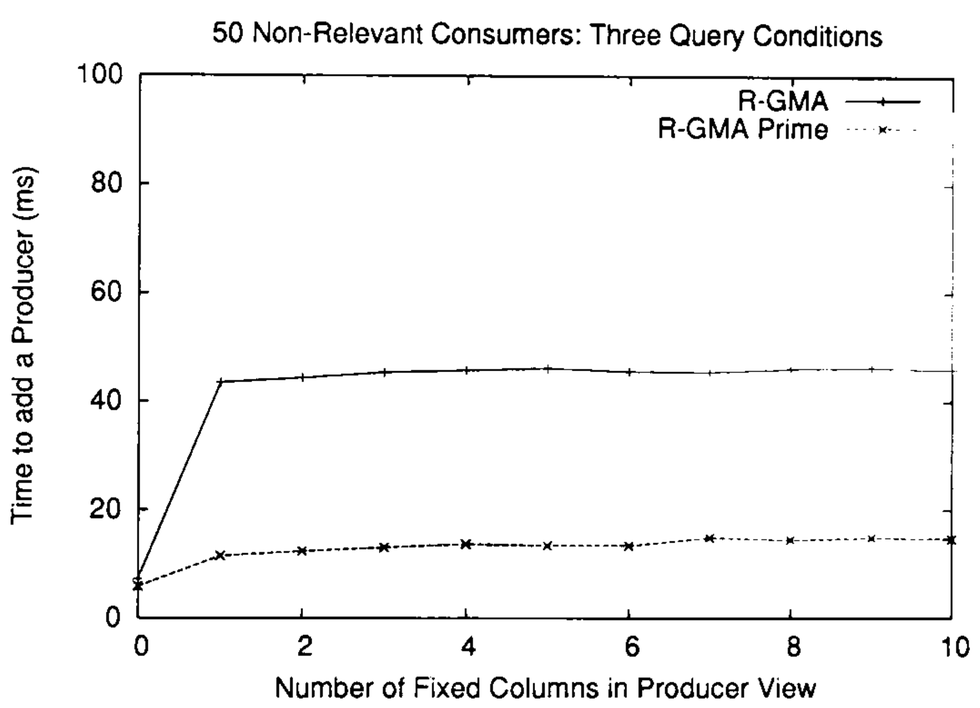
(a) No Consumer predicate.



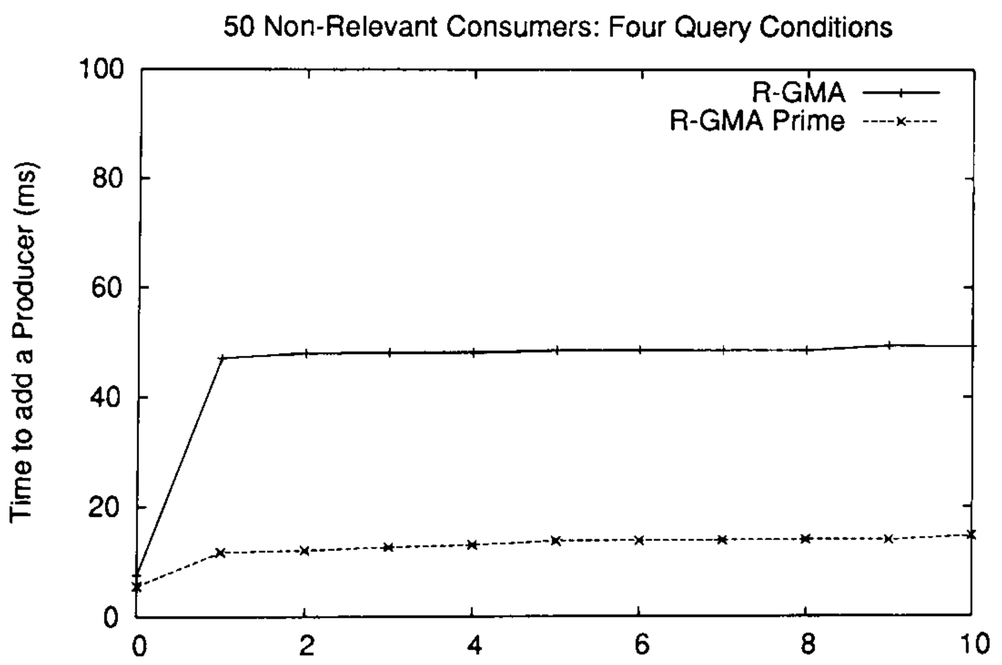
(b) One condition.



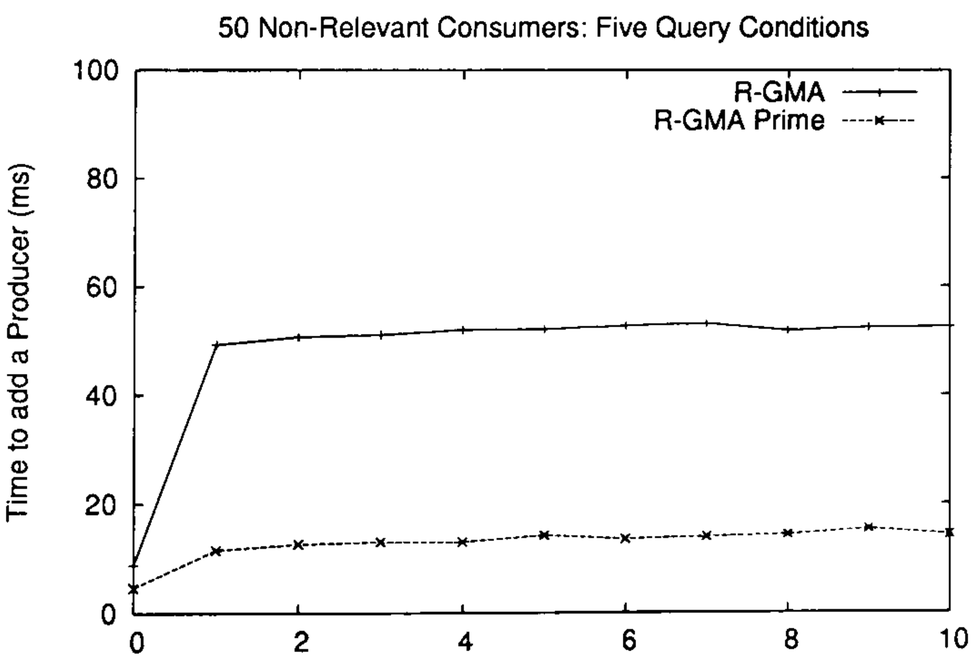
(c) Two conditions.



(d) Three conditions.

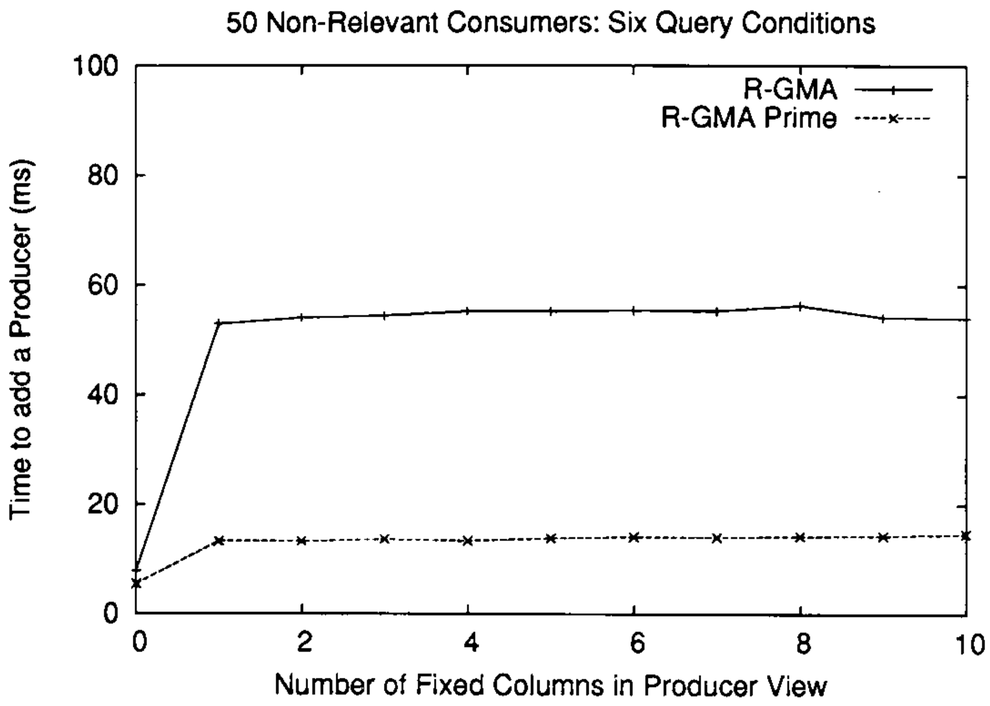


(e) Four conditions.

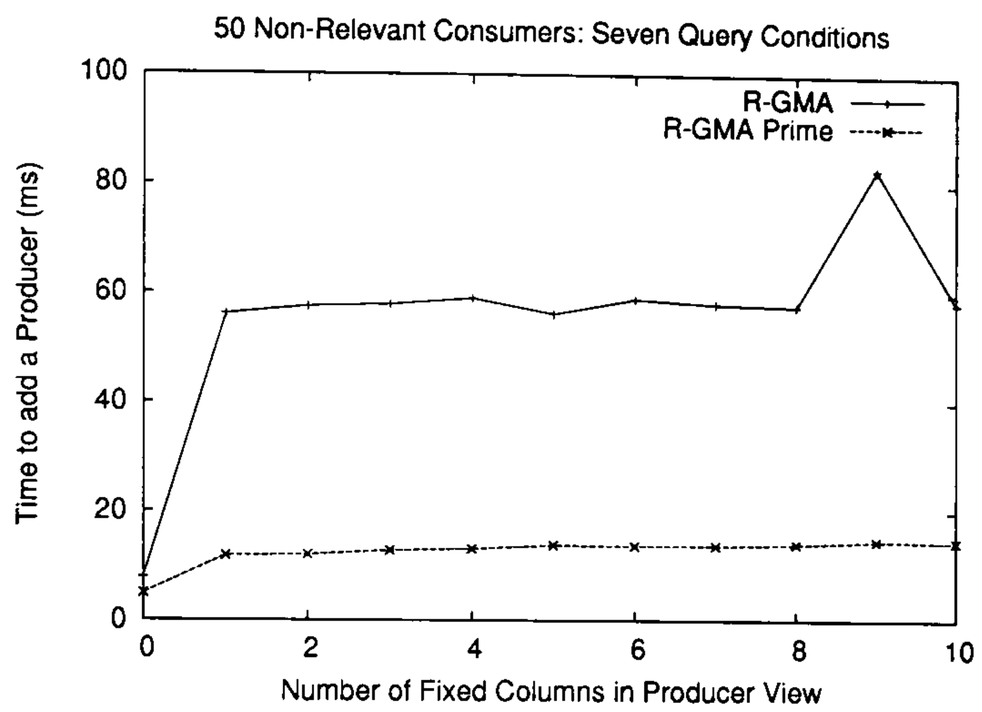


(f) Five conditions.

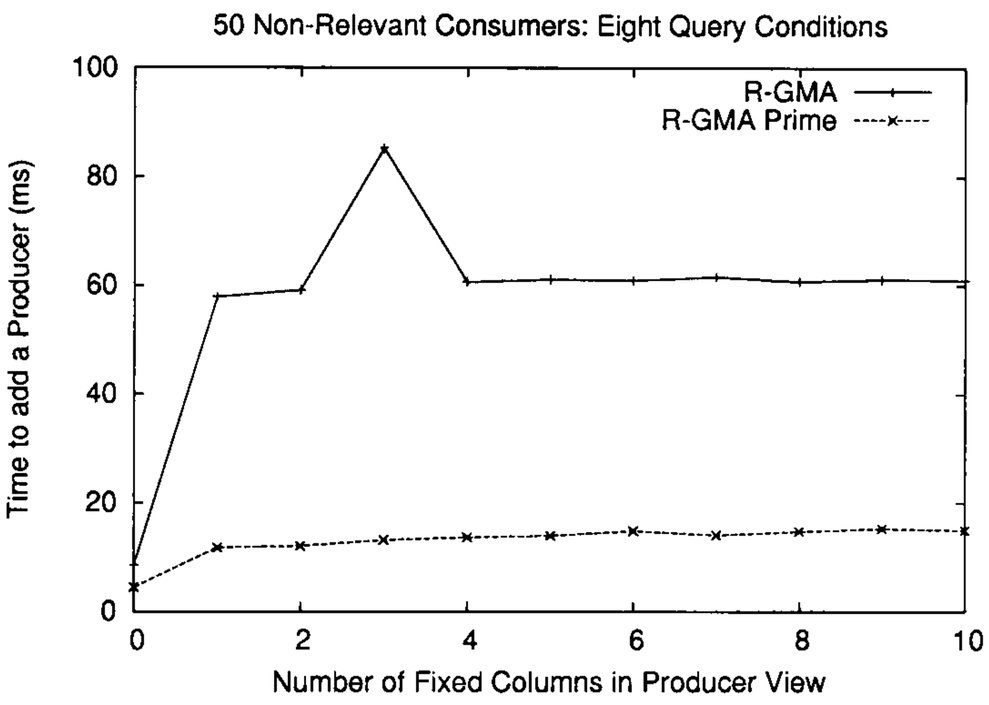
Figure 8.5: Results from registry service performance test with 50 non-relevant consumers, producers added in reverse, i.e. 10 conditions looping down to 0 conditions.



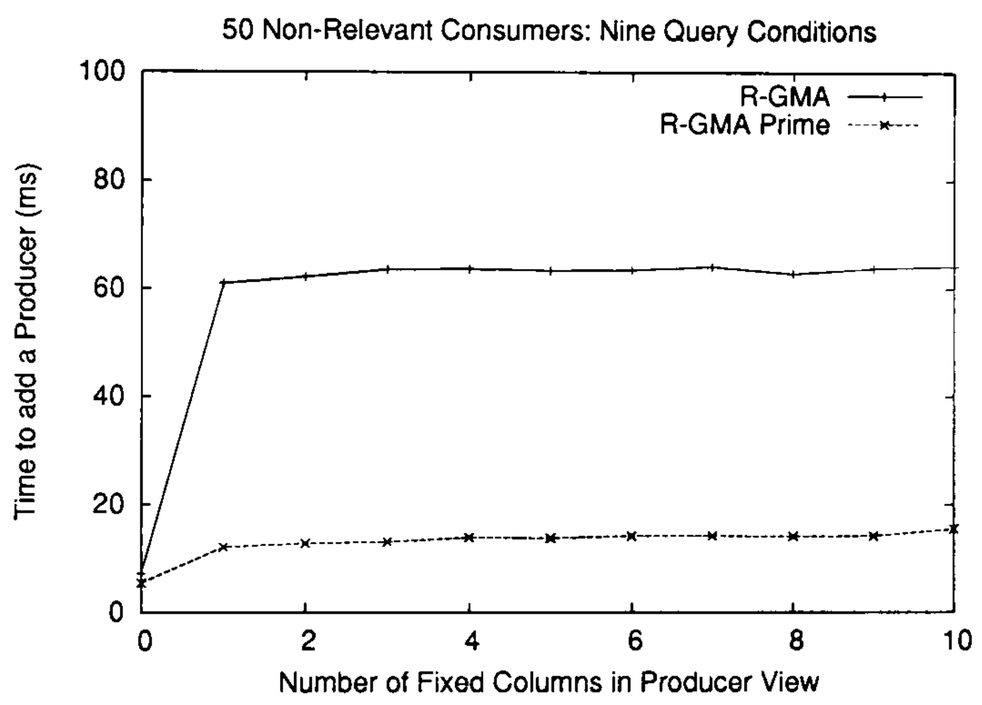
(a) Six conditions.



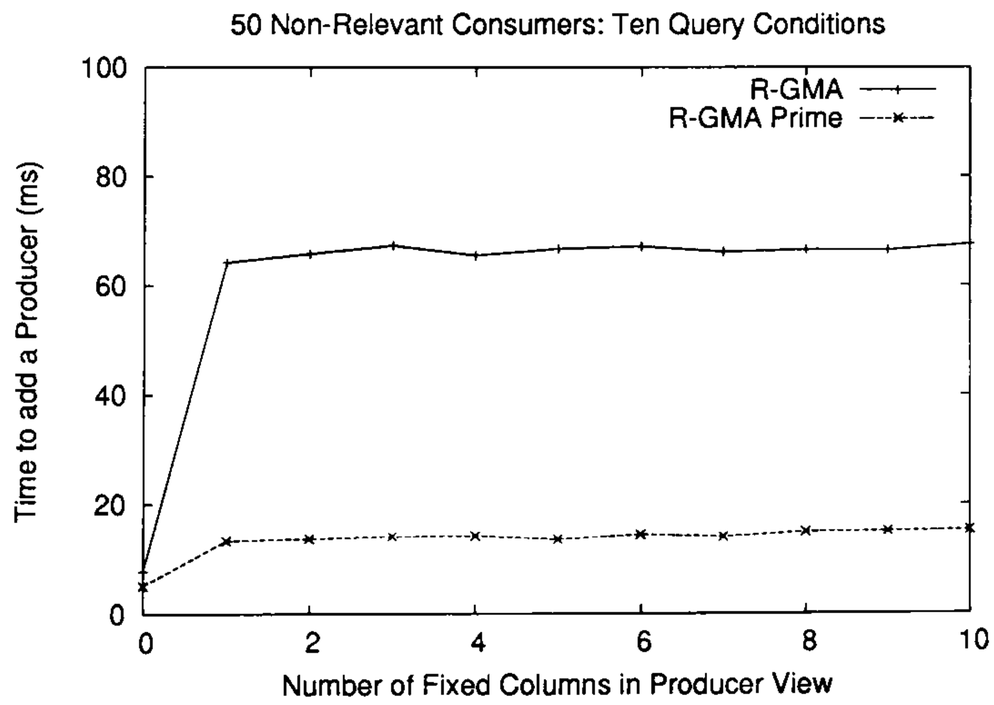
(b) Seven conditions.



(c) Eight conditions.



(d) Nine conditions.



(e) Ten conditions.

Figure 8.6: Results from registry service performance test with 50 non-relevant consumers, producers added in reverse, i.e. 10 conditions looping down to 0 conditions.

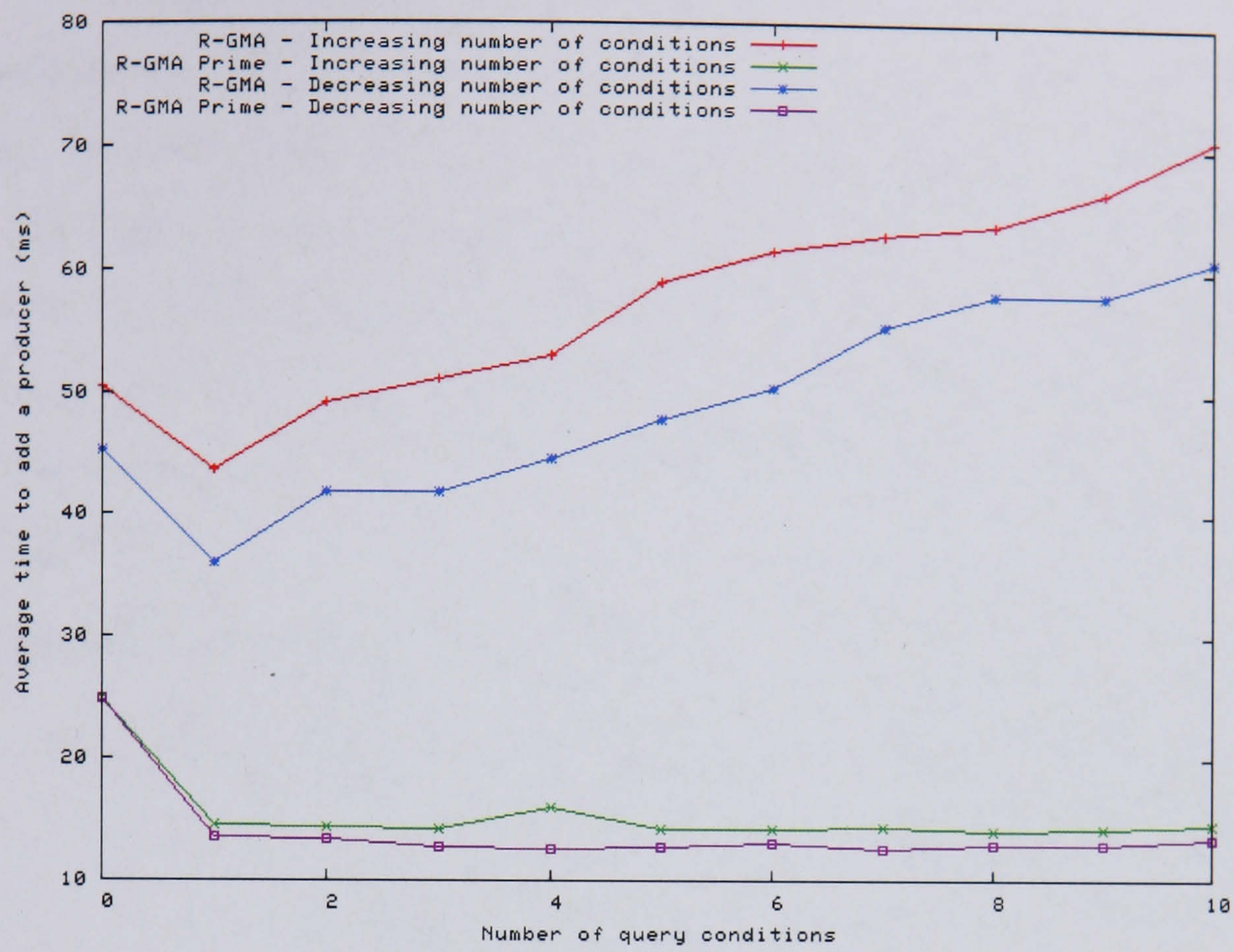
Number of query conditions	R-GMA		R-GMA′	
	Average	Variance	Average	Variance
0	45.3	21.3	24.9	10.1
1	36.1	9.7	13.5	2.7
2	41.8	13.4	13.4	2.5
3	41.9	11.5	12.8	2.6
4	44.6	12.3	12.6	2.5
5	47.9	13.0	12.7	2.9
6	50.4	14.2	13.1	2.6
7	55.5	17.5	12.6	2.7
8	58.0	18.0	12.9	3.1
9	58.0	16.9	12.9	2.7
10	61.0	17.7	13.4	2.8

Table 8.3: The mean and variance for 50 non-relevant consumers run in reverse.

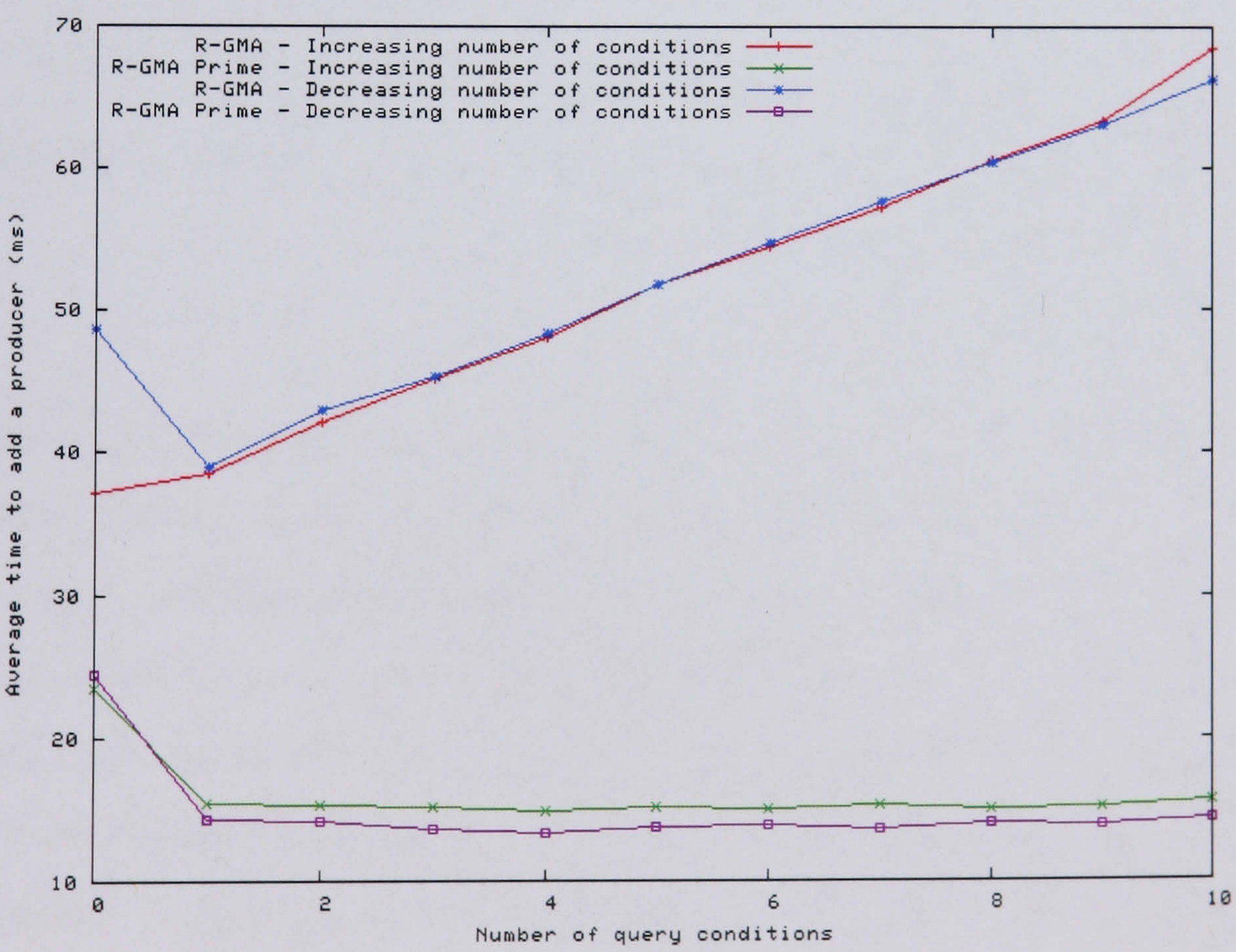
- Figure 8.5(a) shows erratic behaviour throughout.
- Figure 8.5(c) the R-GMA registry service when the views had 5 conditions.
- Figure 8.6(b) the R-GMA registry service when the views had 9 conditions.
- Figure 8.6(c) the R-GMA registry service when the views had 3 conditions.

The spike that was characteristic of running the experiments with an increasing number of conditions is not present when the experiment was reversed. Since the spikes are not present when the view has one or two fixed conditions, it is believed that the spikes are not related to the time taken to perform the relevance test. It is likely that the spikes are caused by the registry service performing some background operation, e.g. the soft-state registration mechanism.

The average and variance of the plots are given in Table 8.3 and are shown in Figure 8.7(a). It is interesting to look at the results for the two runs with 50 non-relevant consumers when the outlying values are removed from the data. These are shown in Figure 8.7(b). Note that the performance of the R-GMA registry service in the two runs is almost identical.



(a) Average time taken to add a producer when all of the consumers are not relevant.



(b) Smoothed average time taken to add a producer when all of the consumers are not relevant.

Figure 8.7: Graphs showing the average time taken to add producers when there are 50 non-relevant consumers.

Experiment with 50 Consumers of Mixed Relevance

The experiments involving 50 consumers of mixed relevance used four different sets of queries. Twelve of the consumers registered the query drawn from the conditions in (8.4) that was always relevant for the producers. Twelve of the consumers registered the query drawn from the conditions in (8.5) that was always not relevant for the producers, except for the case where the consumer has no predicate. Thirteen registered a query drawn from the following 10 conditions, which were relevant for some of the time:

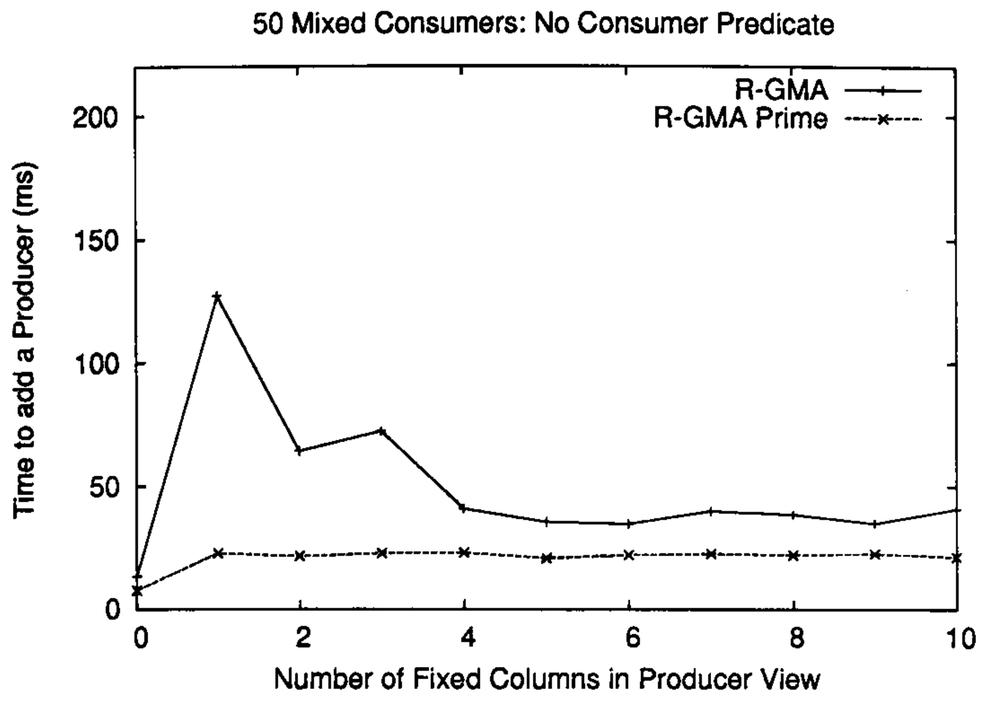
$$\begin{aligned} \mathsf{testColumn1} = \texttt{'constant'} \ \wedge\ \mathsf{testColumn2} \geq 25 \ \wedge\ \mathsf{testColumn3} \leq 66.7 \ \wedge \\ \mathsf{testColumn2} < 50 \ \wedge\ \mathsf{testColumn8} = \texttt{'bbb'} \ \wedge\ \mathsf{testColumn3} > 0.0 \ \wedge \\ \mathsf{testColumn6} = \texttt{'lll'} \ \wedge\ \mathsf{testColumn4} = \texttt{'xxx'} \ \wedge \\ \mathsf{testColumn9} = \texttt{'aaa'} \ \wedge\ \mathsf{testColumn7} = \texttt{'bbb'}. \end{aligned} \tag{8.6}$$

The final thirteen registered a query made from the following 10 conditions, which were relevant for some of the time:

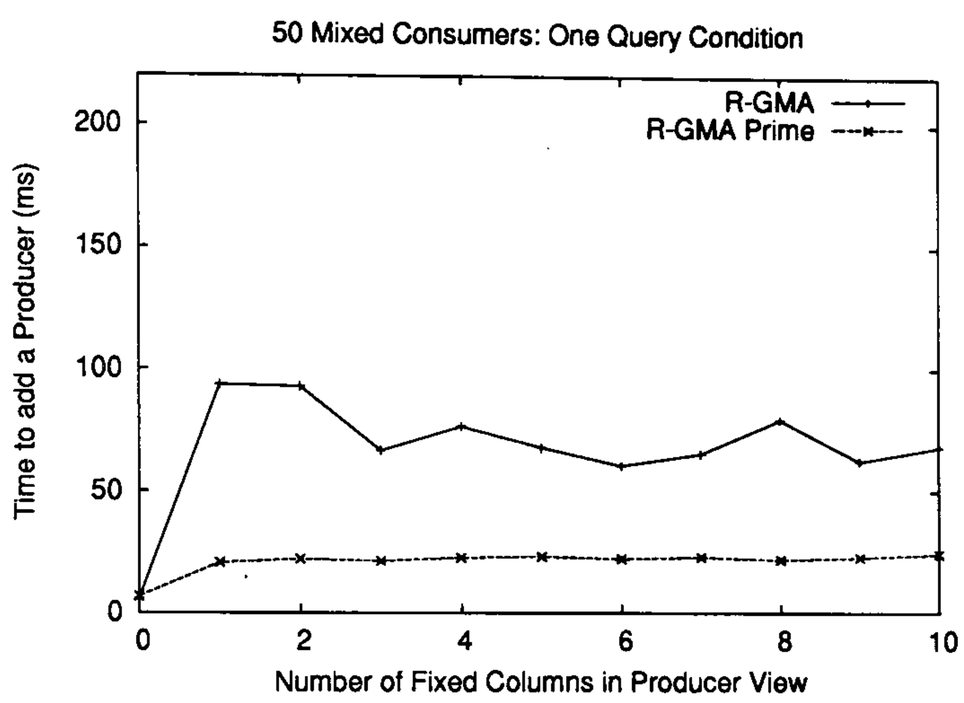
$$\begin{aligned} \mathsf{testColumn2} > 20 \ \wedge\ \mathsf{testColumn3} < 33.3 \ \wedge\ \mathsf{testColumn4} = \texttt{'xxx'} \ \wedge \\ \mathsf{testColumn5} = \texttt{'yyy'} \ \wedge\ \mathsf{testColumn8} = \texttt{'eee'} \ \wedge\ \mathsf{testColumn6} = \texttt{'zzz'} \ \wedge \\ \mathsf{testColumn9} = \texttt{'ddd'} \ \wedge\ \mathsf{testColumn2} \leq 30 \ \wedge \\ \mathsf{testColumn10} = \texttt{'www'} \ \wedge\ \mathsf{testColumn3} \geq 5.0. \end{aligned} \tag{8.7}$$

Figures 8.8 and 8.9 present the results for these experiments. Again, the results show that R-GMA′ registry service outperforms the R-GMA registry service.

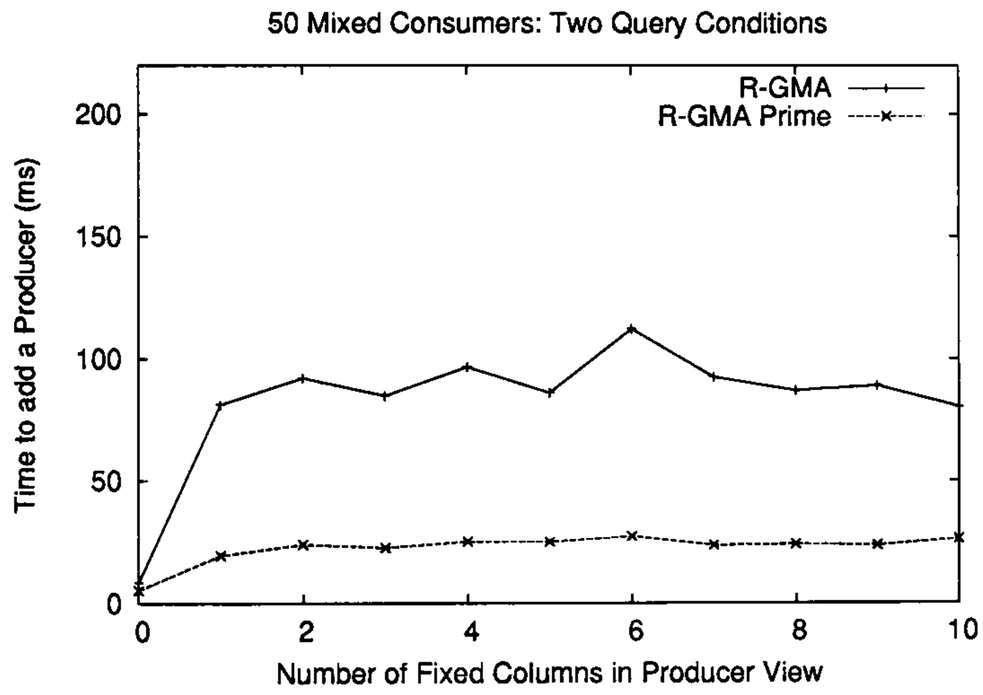
The R-GMA′ registry service shows almost constant performance. The performance of the R-GMA registry service is more difficult to characterise. However, generally the R-GMA registry service took longer to identify the relevant consumers in the first few cases, where the producers only have a few conditions, and then the time taken dropped. In the cases where the consumers have five or more conditions, this drops to almost constant performance, although the point at which this occurs varies. These results are not unexpected. In the first few cases the performance is similar to the experiments where there were 50 relevant consumers. Once the performance starts to drop to a constant value, this mirrors the case when there are 50 non-relevant consumers. An analysis of the query and view conditions shows that the



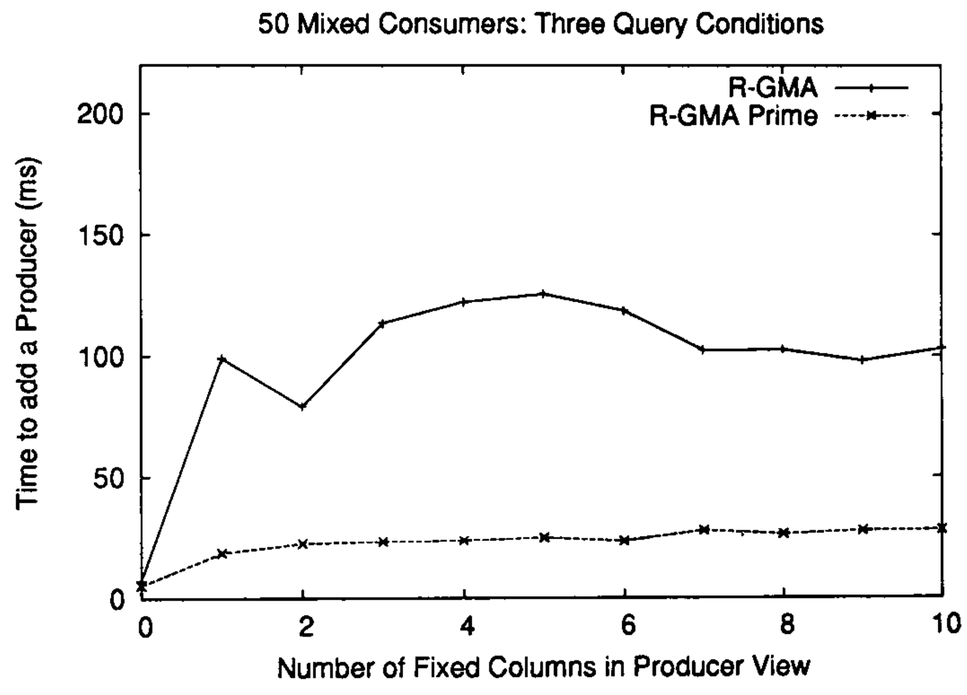
(a) No consumer predicate.



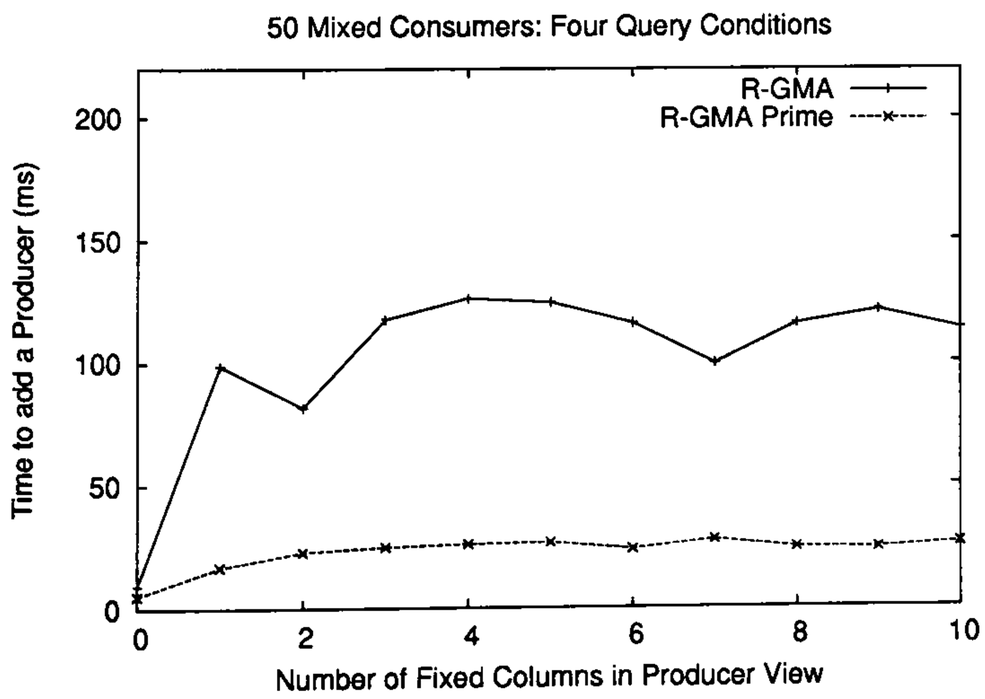
(b) One condition.



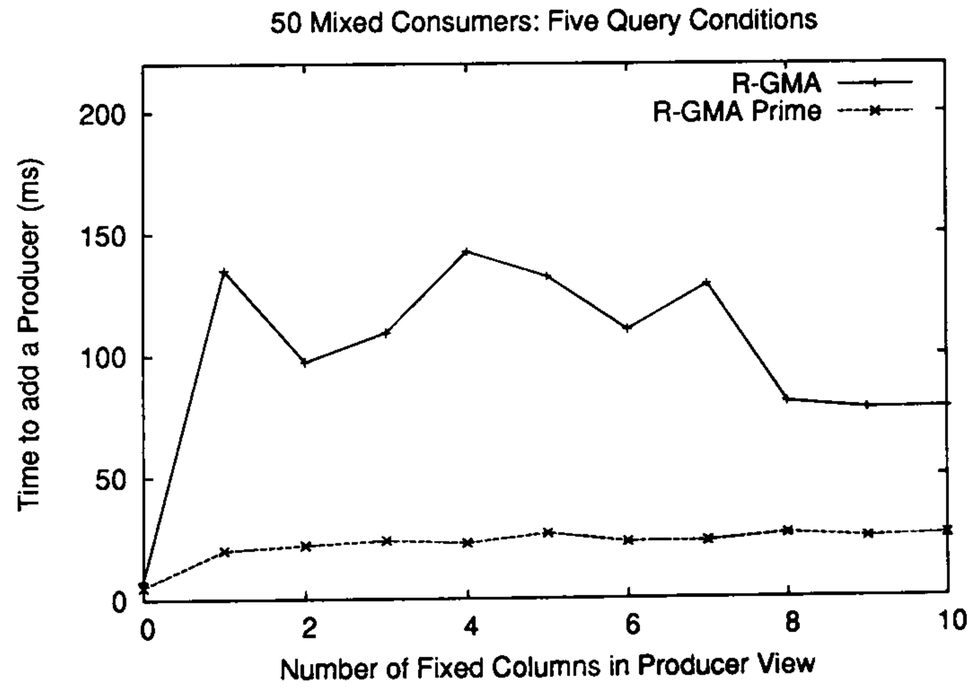
(c) Two conditions.



(d) Three conditions.

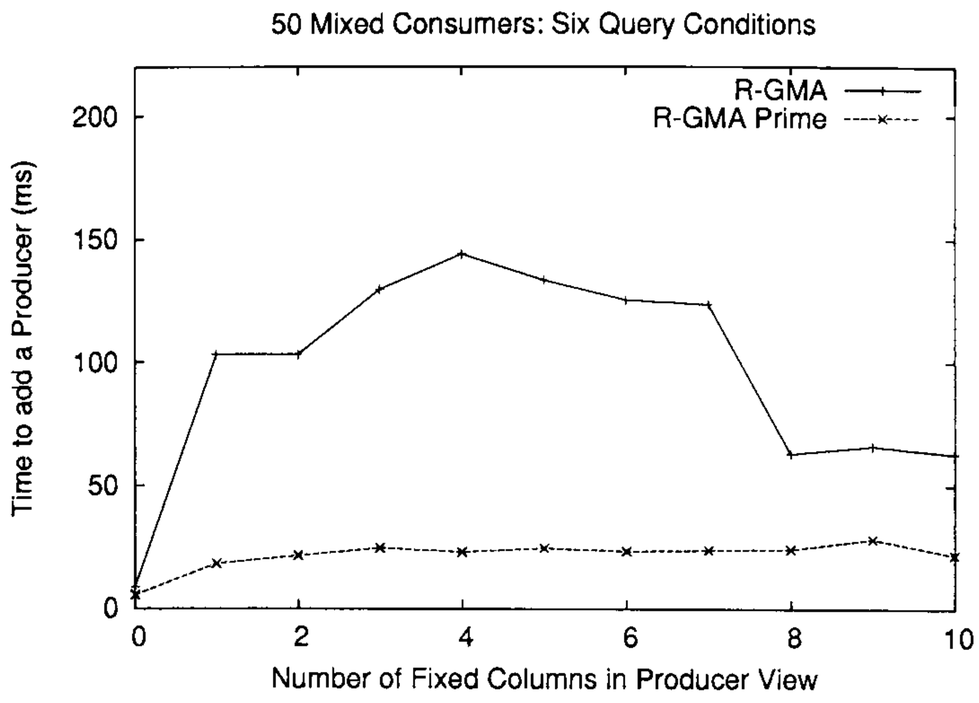


(e) Four conditions.

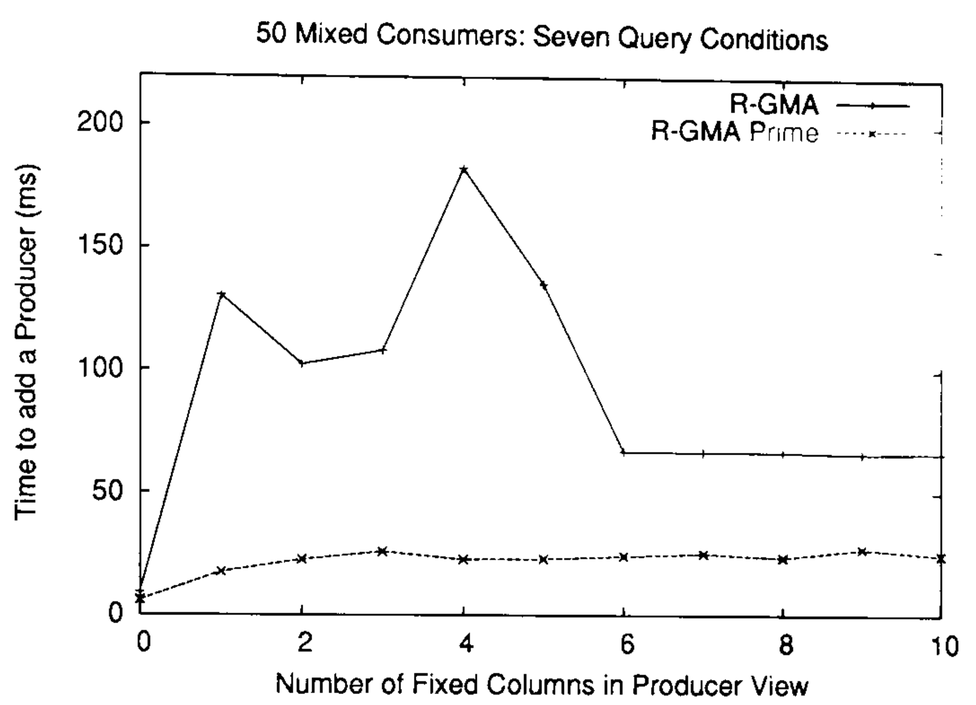


(f) Five conditions.

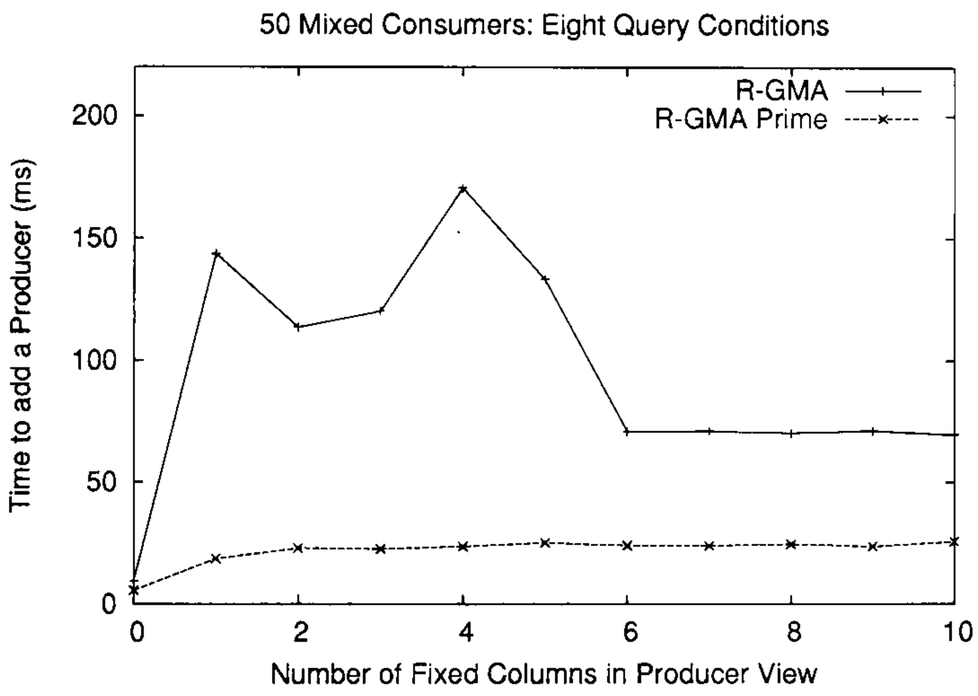
Figure 8.8: Results from registry service performance test with 50 consumers of mixed relevance.



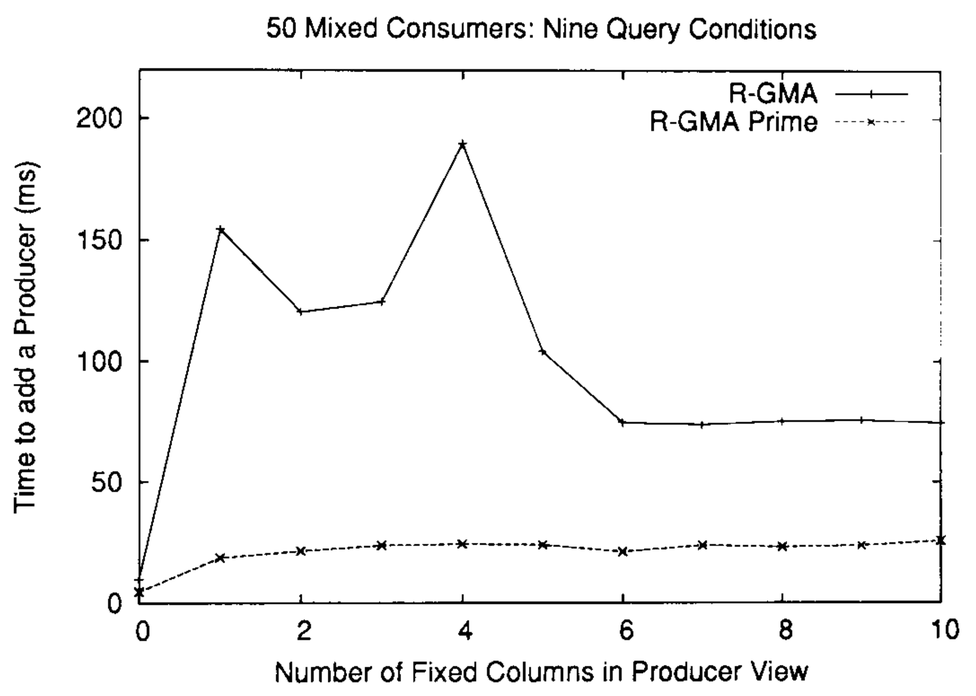
(a) Six conditions.



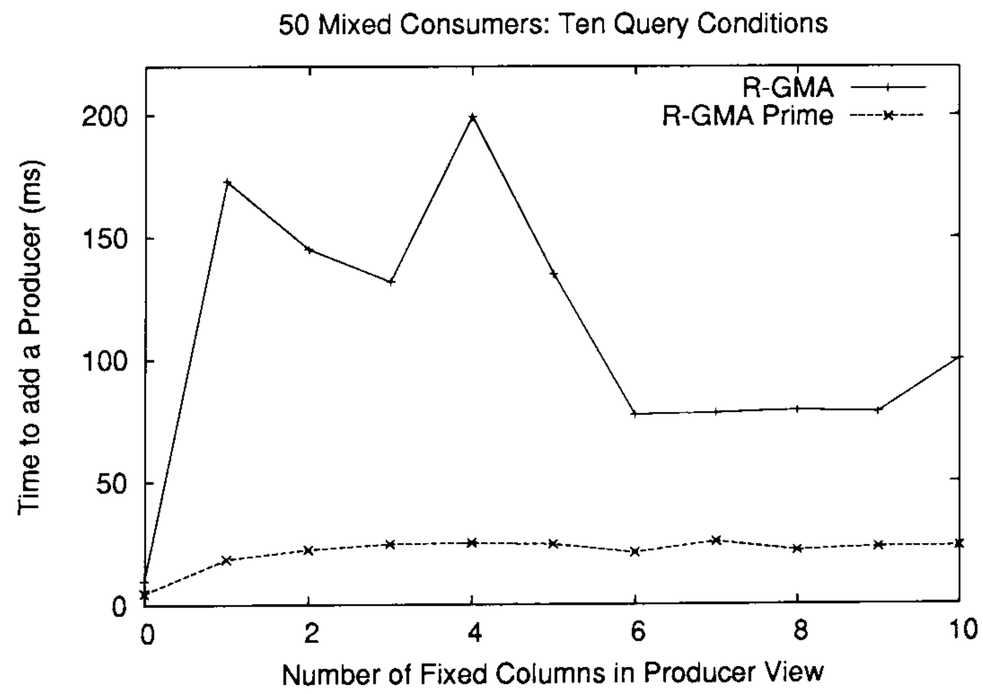
(b) Seven conditions.



(c) Eight conditions.



(d) Nine conditions.



(e) Ten conditions.

Figure 8.9: Results from registry service performance test with 50 consumers of mixed relevance.

Number of query conditions	R-GMA		R-GMA′	
	Average	Variance	Average	Variance
0	49.2	30.1	20.8	4.4
1	67.1	23.3	20.9	4.8
2	82.7	26.1	22.4	6.0
3	97.3	32.6	22.9	6.5
4	102.0	33.5	22.4	6.4
5	99.8	38.8	22.0	6.0
6	96.6	41.5	21.7	5.9
7	91.0	46.9	22.0	5.9
8	94.8	45.6	21.8	5.7
9	97.8	48.3	21.3	5.9
10	109.7	53.3	21.3	5.9

Table 8.4: The mean and variance for 50 consumers with a mixture of queries.

point at which the performance characteristics change is dependent on when each of the Queries (8.6) and (8.7) stop being relevant for the views of the producers. For example, the conditions in Query (8.6) when compared with the view conditions means that the thirteen consumers registering their query based on this query will no longer be relevant when they register seven conditions. This is because the condition

$$\mathsf{testColumn6} = \texttt{'111'} \ \wedge \ \mathsf{testColumn6} = \texttt{'zzz'}, \tag{8.8}$$

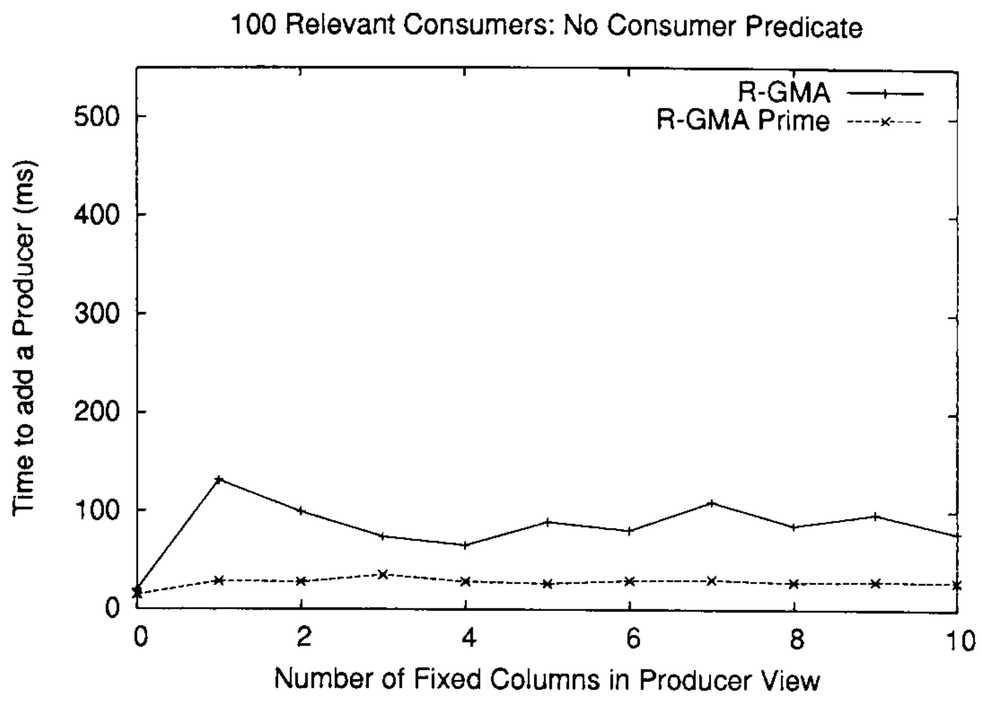
is not satisfiable. In Figure 8.9(b) this is shown by the levelling out of the graph when there are six conditions in the view of the producers.

The average and variance of the plots are given in Table 8.4 and are shown in Figure 8.12(b).

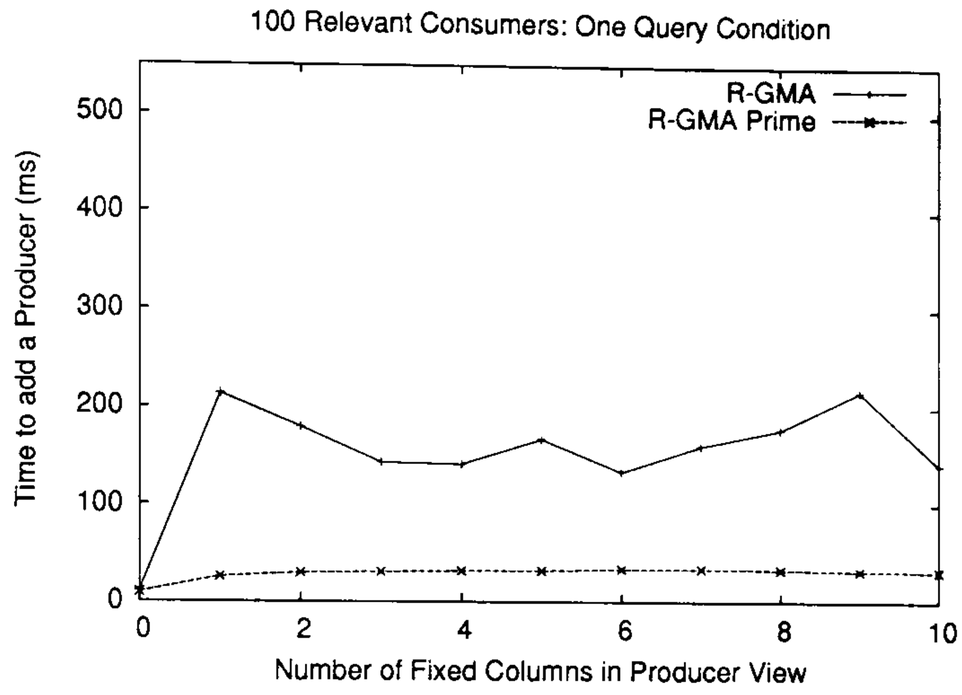
Experiment with 100 Relevant Consumers

The conditions used for the consumers in this experiment were the same as those for the 50 relevant consumers given in (8.4). Figures 8.10 and 8.11 present the results for these experiments.

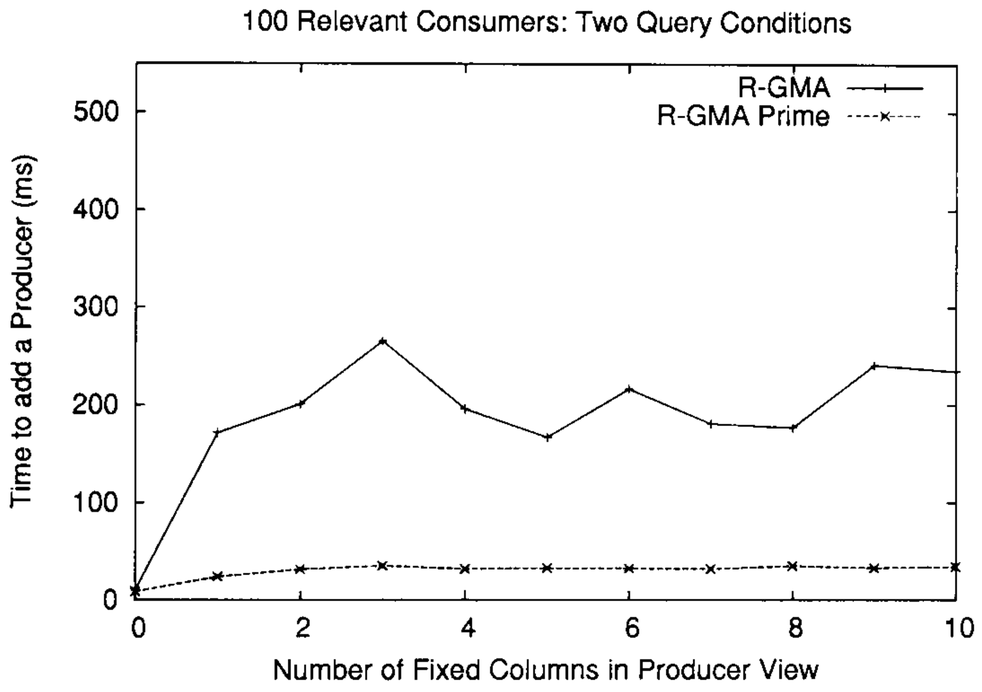
The results are broadly similar to those for the case when there were 50 relevant



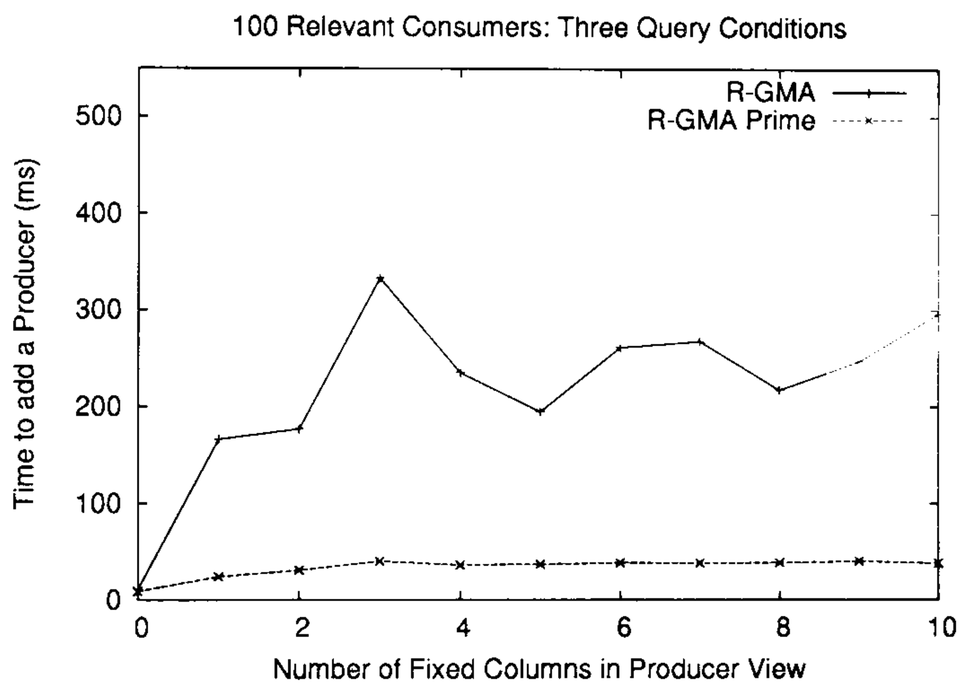
(a) No consumer predicate.



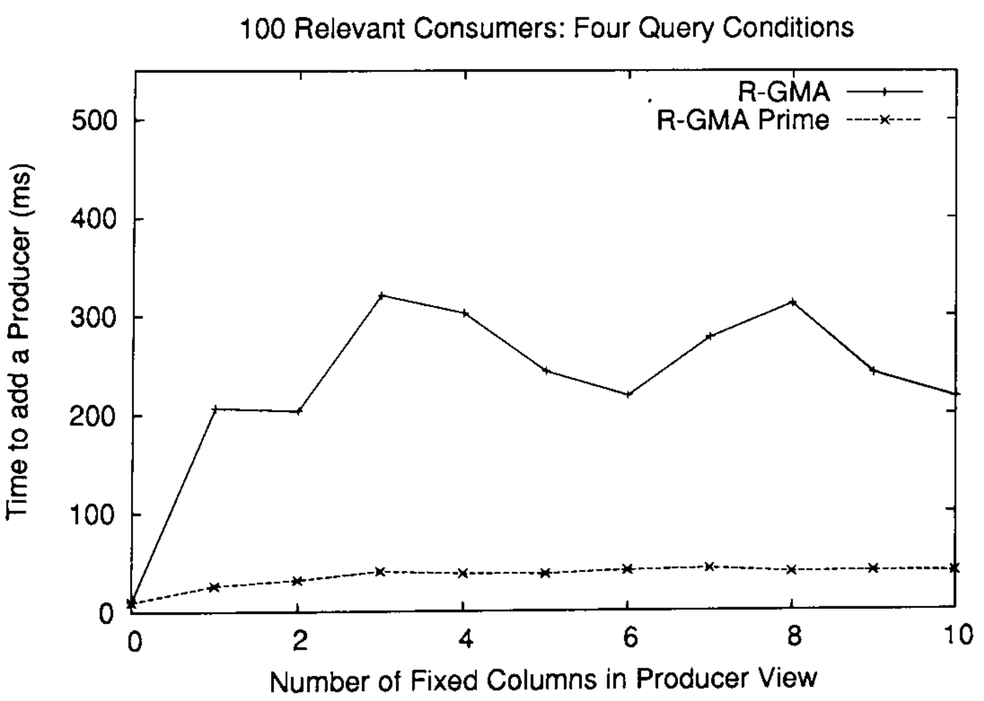
(b) One condition.



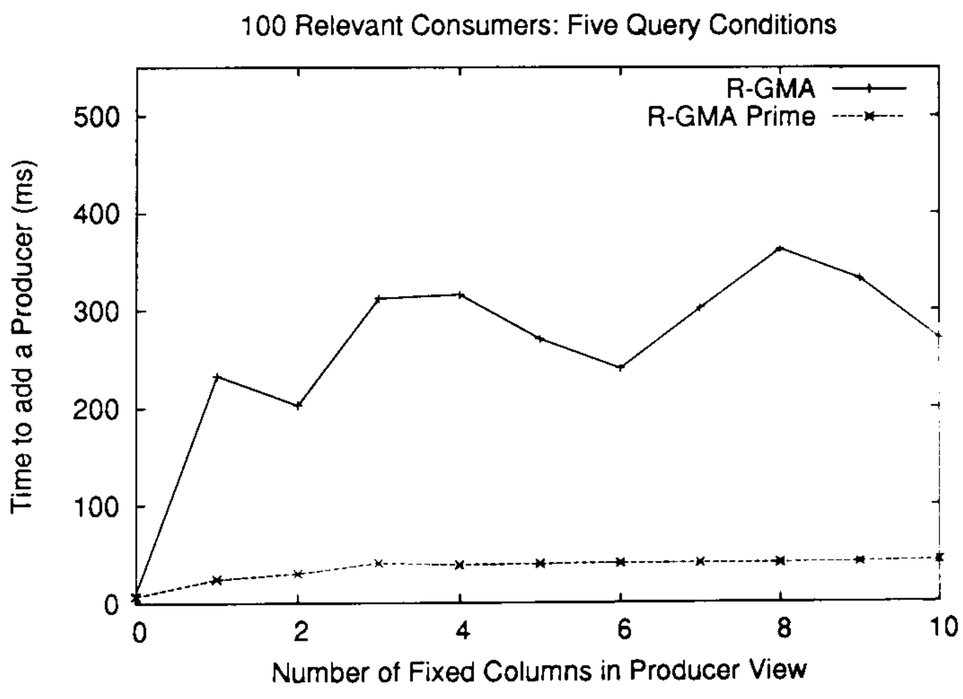
(c) Two conditions.



(d) Three conditions.

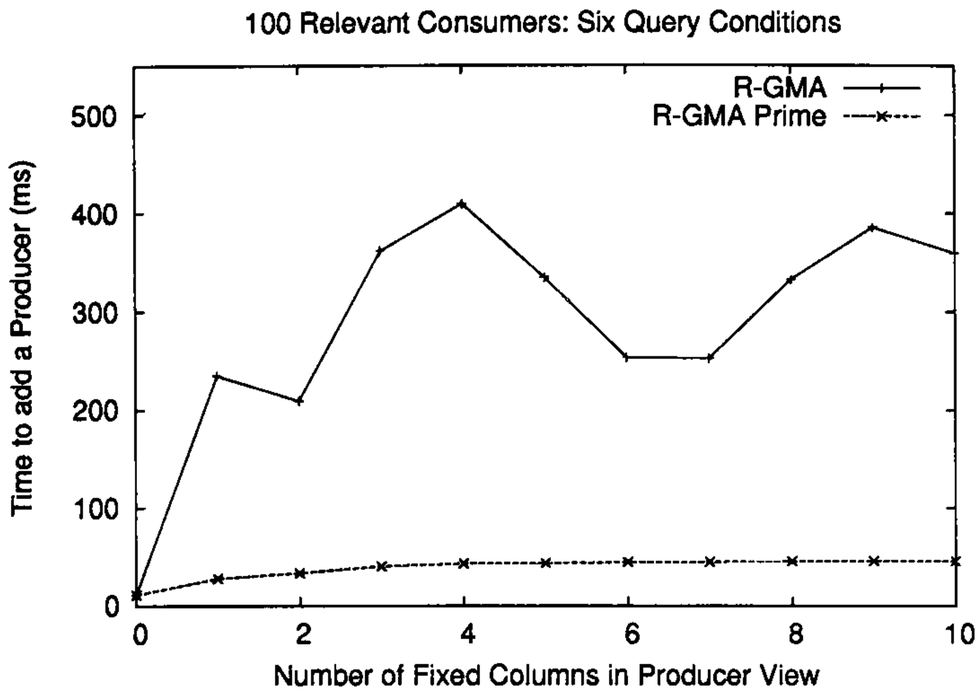


(e) Four conditions.

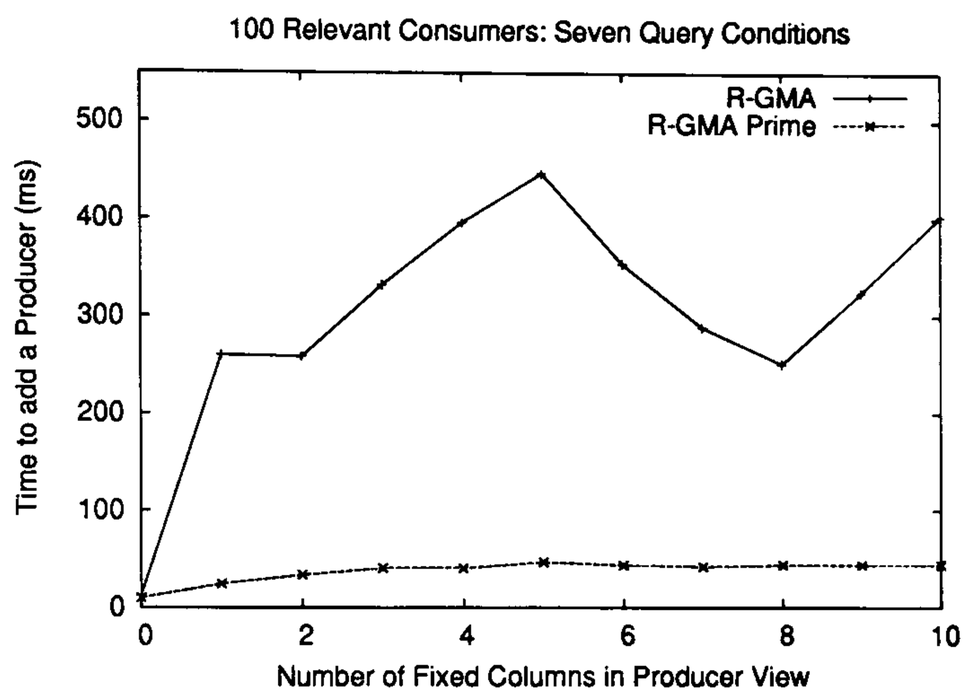


(f) Five conditions.

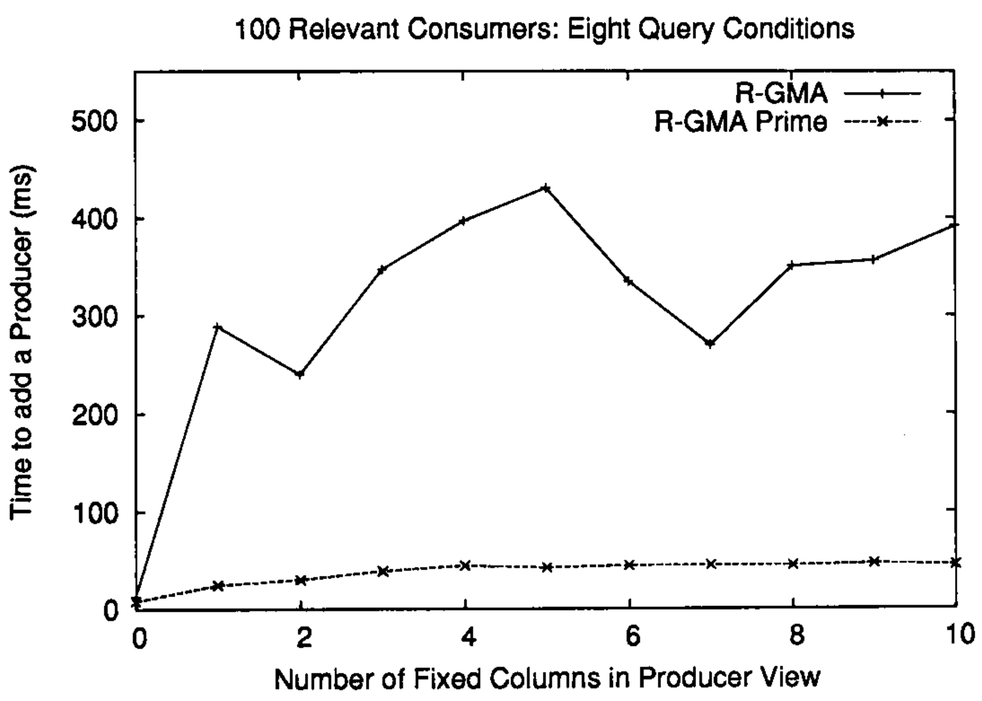
Figure 8.10: Results from registry service performance test with 100 relevant consumers.



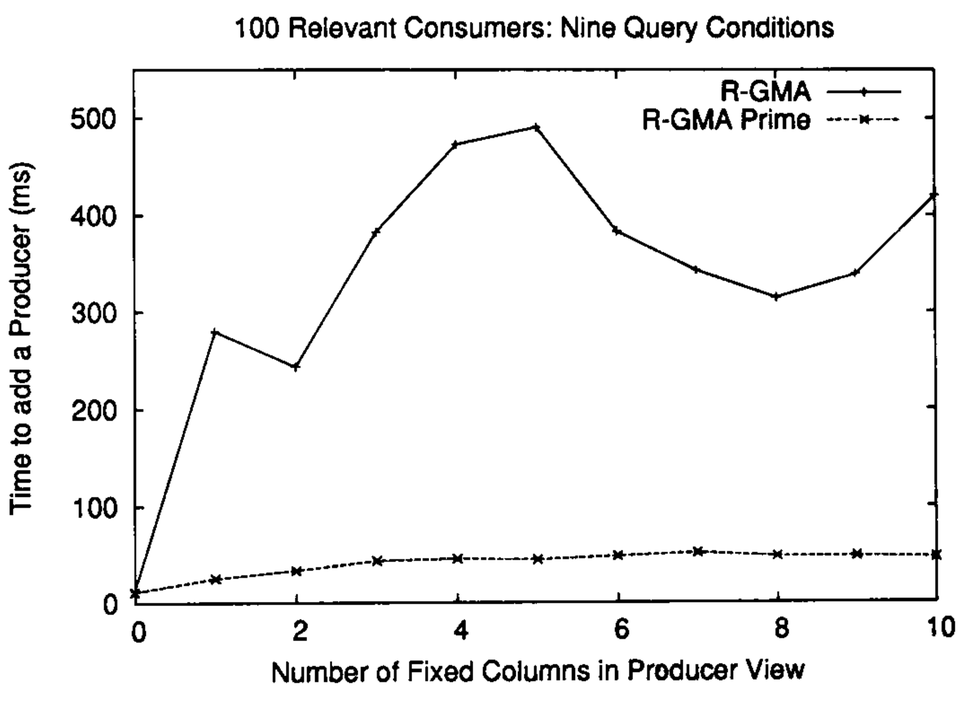
(a) Six conditions.



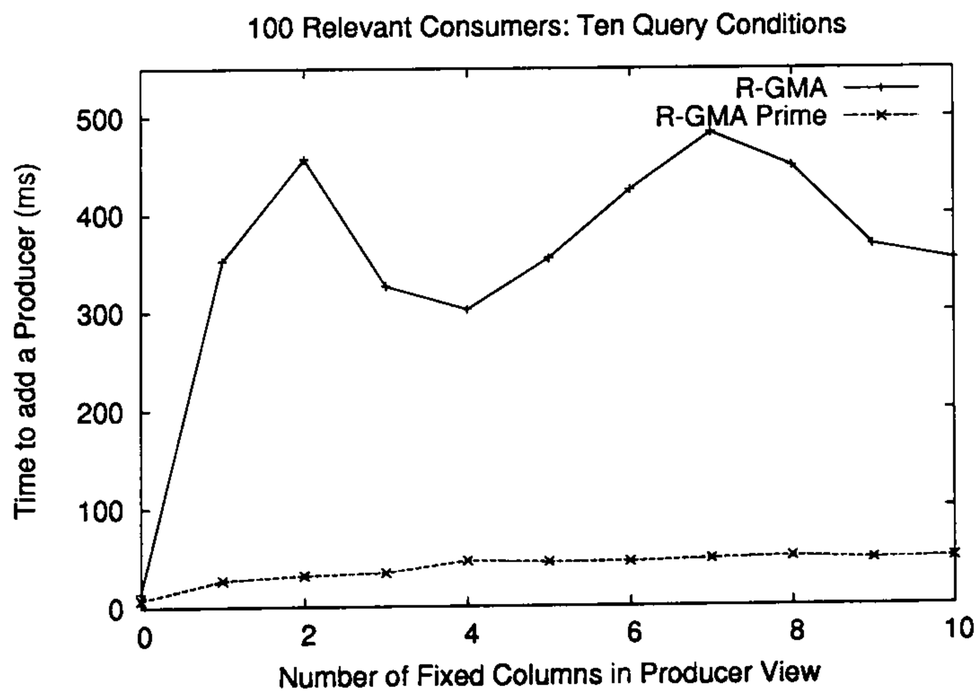
(b) Seven conditions.



(c) Eight conditions.



(d) Nine conditions.



(e) Ten conditions.

Figure 8.11: Results from registry service performance test with 100 relevant consumers.

Number of query conditions	R-GMA		R-GMA′	
	Average	Variance	Average	Variance
0	84.2	28.1	27.4	4.9
1	152.7	55.0	29.1	7.0
2	187.2	66.8	29.9	7.7
3	218.6	85.3	33.5	9.5
4	232.0	85.2	34.9	9.6
5	258.9	95.3	34.9	10.9
6	285.7	112.7	38.5	10.7
7	302.2	116.0	37.4	11.1
8	311.0	113.9	37.6	12.0
9	334.7	130.6	40.4	12.3
10	353.0	126.8	38.7	13.1

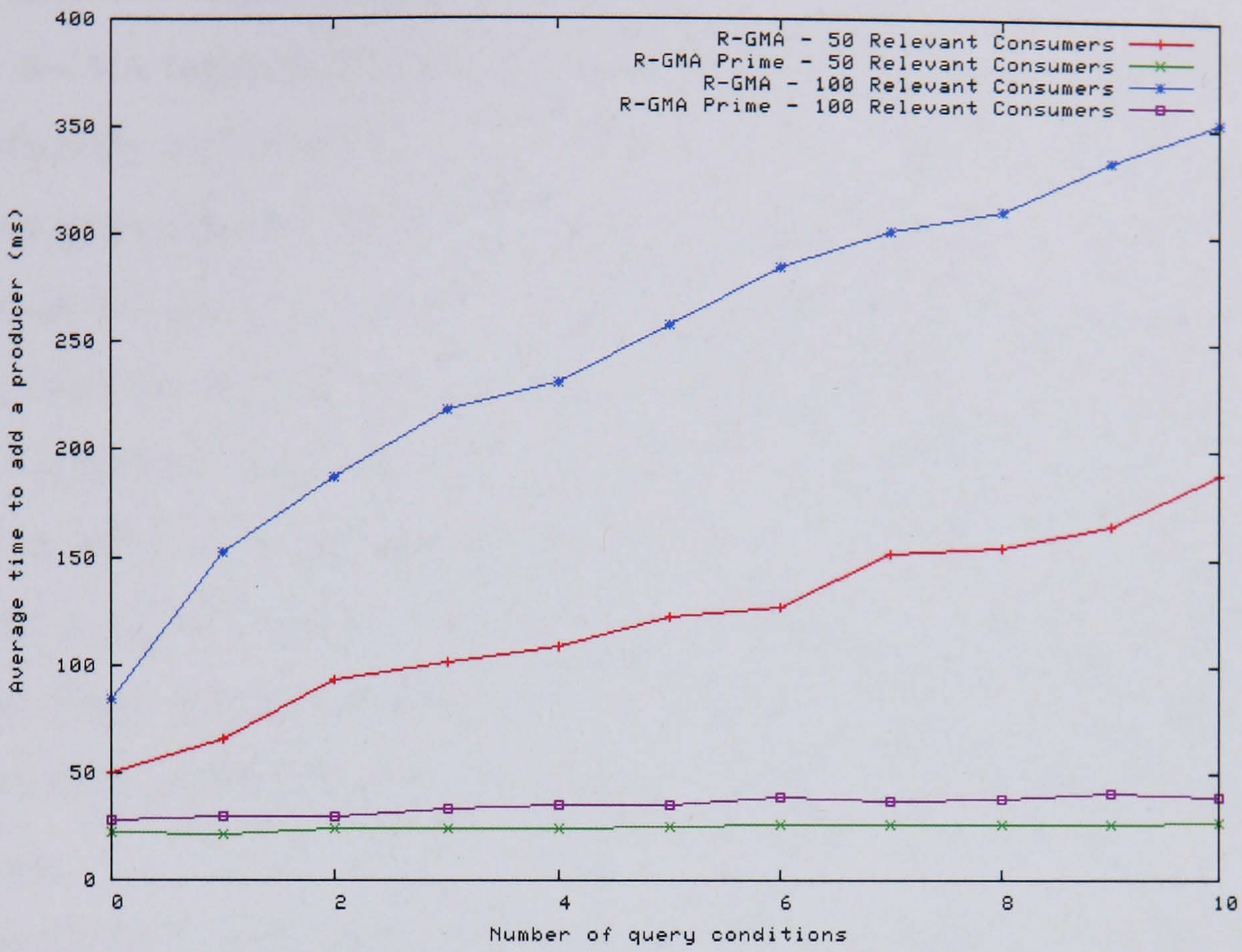
Table 8.5: The mean and variance for 100 relevant consumers.

consumers, although there is more variability in the results. The results show that the time taken by the R-GMA′ registry service to perform the relevance test increases slightly as the number of conditions in the views of the producers increases, although the overall performance is still near constant. For the R-GMA registry service there was a lot of variability in the performance but the trend is an increase in the time taken as the number of conditions increases.

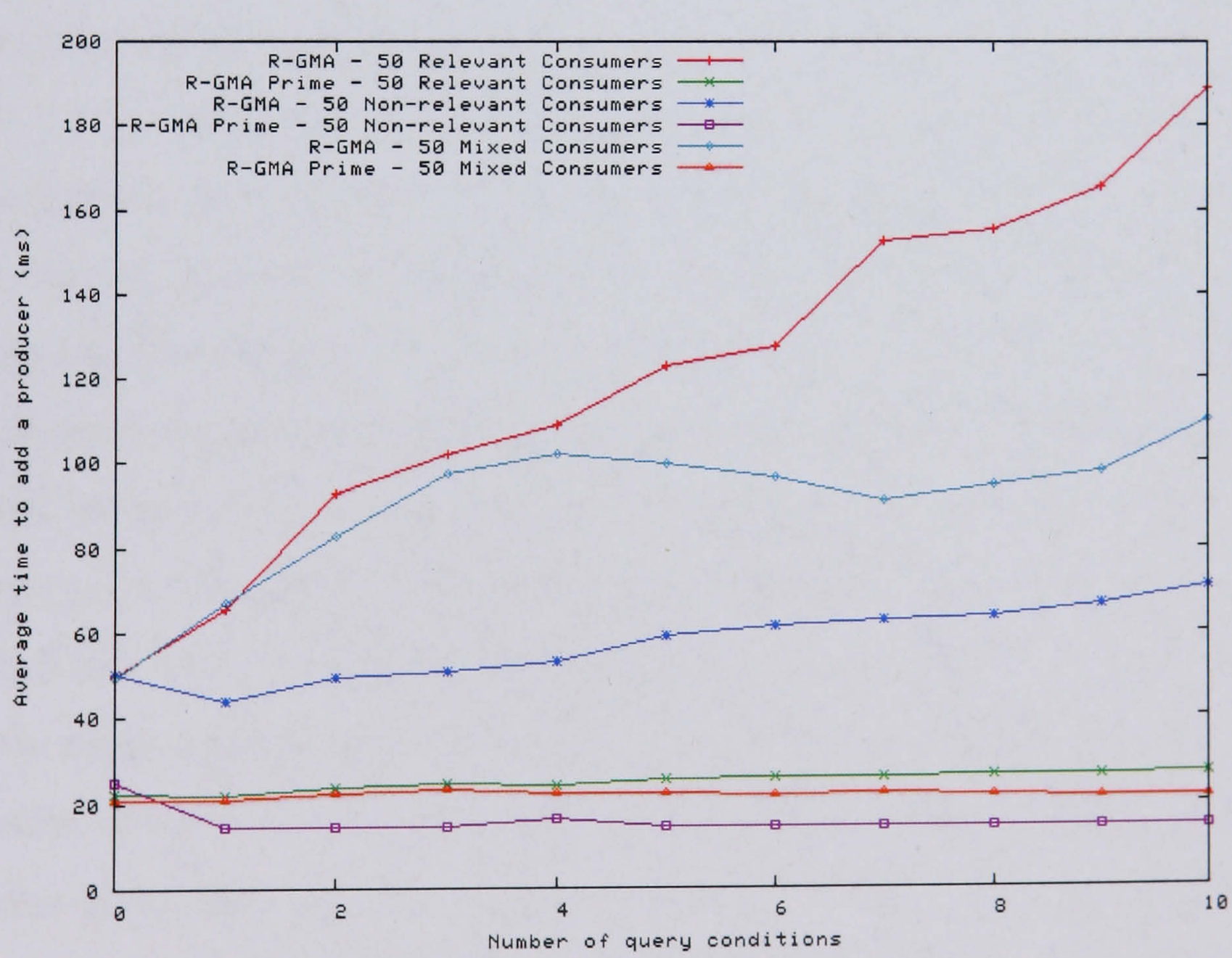
The average and variance of the plots are given in Table 8.5 and are shown in Figure 8.12(a).

8.1.4 Discussion

Overall the results show that there is a significant advantage in storing the queries of consumers in a structured manner in a database and to perform the relevance test as an SQL query. A comparison of the average time taken by the registry services for the cases when there are 50 and 100 relevant consumers is presented in Figure 8.12(a). and shows that for both registries there is an increase in the time taken. For the R-GMA′ registry service this increase is only between 5 and 15 ms. whereas for the R-GMA registry service the performance for 100 consumers is about twice that for 50



(a) Average time taken to add a producer when all of the consumers are relevant.



(b) Average time taken to add a producer when there are 50 consumers registered.

Figure 8.12: Graphs showing the average time taken to add a producer as the number of conditions in the queries increases.

consumers.

This dramatic improvement in the performance can be accounted for by the fact that the R-GMA registry service must parse each consumer query before performing the satisfiability test whereas the improved registry service R-GMA′ can exploit the structured representation of the query condition in the registry service database.

It should be noted that the results for the R-GMA′ registry service did not include the time taken to initially parse and store the queries of the consumers when they were first registered. However, the additional time taken for this process is only that required to store the conditions in the database since the current R-GMA registry service must already parse the query to check that it is a permissible continuous query, i.e. it is a valid selection query over one of the tables in the schema. Thus, the additional time taken is only that of generating and executing a handful of SQL insert statements.

It was claimed in Section 8.1.2 that the three experimental conditions involving 50 consumers would model the performance for the best-case, worst-case, and an approximation of the normal-case. The averages for these experiments are presented in Figure 8.12(b) and clearly show that the results for the experiment involving 50 mixed consumers lie between the other two experimental conditions. Notice that for both registry services the experiment involving 50 mixed consumers begins by resembling the performance for when all the consumers are relevant and then as the number of query conditions increases more closely resembles the performance for the experiment when all of the consumers are not relevant. This feature is likely to be caused by the choice of query conditions for the consumers in the experiment.

Most of the experiments showed a spike at the start of each run, e.g. Figure 8.3(b). A possible cause for the spike could be the effect of the MySQL database management system making use of a data cache. The effects of this were not investigated. Firstly, this is because any such cache would affect each registry service. Secondly, the purpose for conducting the experiments was to investigate which registry service was more efficient. All of the results show that the R-GMA′ registry service approach of using a structured representation significantly outperformed the R-GMA registry service approach of storing the consumer conditions as a string.

There are a number of other advantages in using a structured representation for

the queries of the consumers. Firstly, there would no longer be a limit on the length of the `WHERE` clause in the query. This limit has not yet been reached in the deployment of the R-GMA system but currently the predicate can only be 255 characters long[1].

Secondly, it allows for more code reuse in the implementation of the registry service. In the current R-GMA implementation, there are separate methods for handling the views of the producers and the queries of the consumers. By storing the information about both producers and consumers in the same way, common methods can be used to process the information. Similarly, when performing the relevance test, the same code can be used to generate the sub-queries that perform the satisfiability test, see Section 7.3.2.

Finally, it allows republishers to pose arbitrary selection queries rather than being limited to the views that producers can register, since a common approach can be followed for storing their details. This is necessary if republishers are to be used for answering continuous queries and hence the creation of hierarchies of publishers.

8.2 Effects of a Publisher Hierarchy

The second set of experiments performed for this thesis investigated the effects of a hierarchy of publishers on the latency of an answer tuple. The republishers were introduced to allow continuous queries to be answered more efficiently since the consumer need not contact as many publishers. This should also allow the system to scale to large numbers of producers and consumers. However, the republishers will increase the time taken for a tuple to reach a consumer. This is because the tuple must travel from the producer that publishes it, through some number of republishers before reaching the consumer.

The experiment is designed to quantify the time taken by a republisher to receive a tuple and then make it available in the answer stream. It will also investigate whether there is a linear growth in the latency with additional levels of a publisher hierarchy.

[1]For the experiments the R-GMA registry service database was altered to allow queries with longer predicates to be stored by changing the data type to a text field.

8.2.1 Experimental Method

The experiment was run over the extension of the R-GMA system presented in Section 7.3. The experiment consisted of a producer which publishes a set number of tuples each containing the current time in milliseconds. Between publishing tuples, the producer waited 10 ms. The consumer, upon receiving a result set of tuples, would add a line to the result file for each tuple in the result set containing the time at which the result set was received, in milliseconds, and the time at which the tuple was published. After the experiment was run, the result file was processed so that the time at which the tuple arrived at the consumer was compared against the time when it was published. The difference in these times would be the time taken to deliver the tuple.

8.2.2 Experimental Setup

The experiments were run on six machines, each of which was installed with the extended version of R-GMA. The machines were configured so that one machine acted as both the registry service and schema service. One machine acted as both the producer service and the producer publishing the tuples. Another machine acted as both the consumer service and the consumer. Three machines were used to host one republisher each along with the appropriate services.

The experiment was initially conducted with no republisher. This would provide the latency of sending a tuple directly from the producer to the consumer. The experiment was then run with one republisher which consumed from the producer and streamed to the consumer. The next experiment used two republishers with one republisher feeding the other. The final experiment consisted of three levels of republishers between the producer and consumer.

In all of the experiments 10,000 identical tuples, except for the current time, were inserted by the producer. Each experiment was repeated three times and the results averaged.

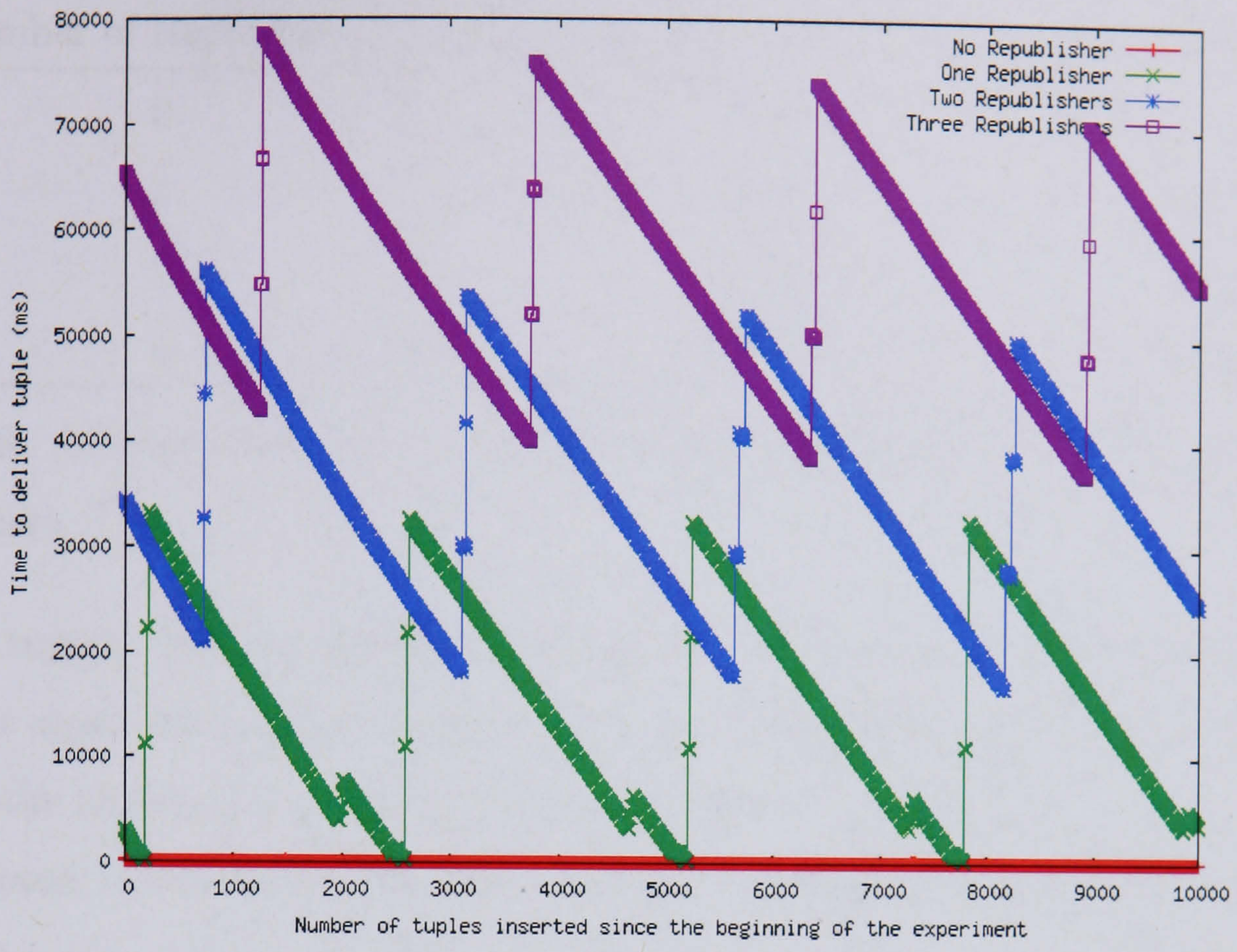


Figure 8.13: Time taken to deliver tuples from the producer to the consumer using different numbers of republishers.

8.2.3 Results

The results for the experiment are presented in Figure 8.13. The experiments clearly show that the time to deliver a tuple directly from the producer to the consumer is constant and takes a few milliseconds. For the republishers, there is an interesting saw-tooth function shape to the graphs. The average time to deliver a tuple in each of the experiments, along with the variance, is shown in Table 8.6.

The saw-tooth function shape to the graphs involving republishers can be explained by some understanding of the implementation of the republishers. A republisher consists of two separate R-GMA components, each with their own agent:

1. A consumer which poses the republisher's query and retrieves the answer stream from the relevant publishers in the query plan, and

2. A producer which publishes the resulting answer stream in the system.

When a republisher is created, it waits 10 seconds before collecting any tuples from its consumer agent. This means that it receives a block of all the tuples so far streamed to the consumer agent. These are then inserted into the republisher's producer as a

Number of Republishers	Average time to deliver a tuple (ms)	Variance
0	26	10
1	15085	9673
2	35300	9947
3	57270	10141

Table 8.6: Average time and variance to deliver a tuple using different numbers of republishers.

block of tuples. The republisher then "sleeps" for 30 seconds before contacting its consumer agent for any new tuples. Thus, the republisher adds up to a 30 second delay to the latency for a tuple and groups tuples up into blocks.

The peak values of the saw-tooth feature, together with the tuple number at which it occurs, are given in Table 8.7. For one republisher the peak value is roughly constant. For two republishers the peak value decreases gradually and for three republishers the peak value decreases at a faster rate. However, it is likely that the peaks of the saw-tooth function are dependent on the synchronisation between a republisher polling its consumer agent and the rate of publication of tuples by the publisher from which it consumes.

An interesting feature of the graph for one republisher in Figure 8.13 is the small peak that occurs before the time taken to deliver a tuple reaches its lowest value in each of the saw-tooth features. This small peak is likely to be caused by the following sequence of events:

1. The republisher awakes and polls its input queue and discovers that it contains tuples.

2. The republisher processes the tuples on its input queue.

3. The republisher polls its input queue. If the queue is not empty then repeat step 2, otherwise sleep 30 seconds and repeat step 1.

In the first step, the republisher will have been asleep for some period of time which leads to a build-up of tuples on its input queue. When it awakes it purges this queue and processes the tuples which takes time but not as long as it was asleep for. When

Tuple Number	One Republisher	Two Republishers	Three Republishers
210 734 1267	33413.33	56316	78484.33
2613 3148 3773	32791.33	53856.67	75968.67
5229 5724 6378	32698.33	52286	74125.67
7817 8253 8937	32740	49908	70476

Table 8.7: Peak values for the republishers.

it re-polls its queue there is a smaller number of tuples. This repeats until there are eventually no tuples to process since it takes time to insert a tuple and the producer is only publishing every 10 ms. The republisher then sleeps for 30 seconds and repeats the process.

The graphs representing the runs with two and three republishers do not show this small spike feature. This is because the small spike feature is caused by the synchronisation between the producer which has a 10 ms sleep period and the republisher which has a 30 s sleep period. However, it is possible that when there are two or more republishers that a suitable synchronisation could result.

Consider the case of two republishers R_1 and R_2, where R_1 consumers from the producer and R_2 consumes from R_1. Republisher R_1 will go through the sequence of events outlined above. The graphs in Figure 8.13 show the case where R_2 wakes when R_1 has recently entered a sleep 30 s phase. This means that no tuples arrive at the consumer agent of R_2 whilst it is processing the tuples that have arrived at its consumer agent whilst it was sleeping. Thus, R_2 goes straight into a sleep 30 s phase. However, if R_2 started processing its tuples when R_1 was in a small spike phase then tuples would arrive at the consumer agent of R_2 and when it finished processing its

Number of Republishers	Average time to deliver a tuple (ms)	Variance
0	527	236
1	15462	9631
2	35572	10107
3	57213	10135

Table 8.8: Average time and variance to deliver a tuple using different numbers of republishers when the producer introduces no delay between inserting each tuple.

initial batch of tuples there would be more for it to process. This would result in a small spike in the graph with two republishers, although it is highly dependent on the synchronisation of all of the sleep periods.

The average values in Table 8.6 show that one level of republishers introduces a delay of 15 seconds, a second level of republishers adds 20 seconds to the delay, and the third level adds another 22 seconds to the delay. The same experiment was also conducted with the producer adding no delay between inserting each tuple. The results for this experiment are shown in Figure 8.14. The graphs in Figure 8.14 show the same characteristics as the graphs in Figure 8.13. The average values for this run are shown in Table 8.8. Note that the average values are similar to those in Table 8.6, except for the case where there is no republisher which has a large variance value in Table 8.8. This shows that the synchronisation between the publishers is stable. However, in both experimental runs the republishers were created one after the other and, since the dominant delay is from the republisher sleep period, are likely to show the same synchronisation patterns between their sleep periods. An experiment where the republishers were not started one after the other or where the each of the republishers slept for different amounts of time could show different average delay times.

Currently, the delay of 30 seconds is hard coded into the R-GMA republisher. However, there have been requests from users of the R-GMA system that the delay should become a configurable parameter. It is anticipated for a future release of R-GMA that the administrators of the republishers will be able configure the delay between each round of polling. Thus, the experiments were repeated but with the republisher introducing no delay. i.e. the republisher continuously polls its agent for

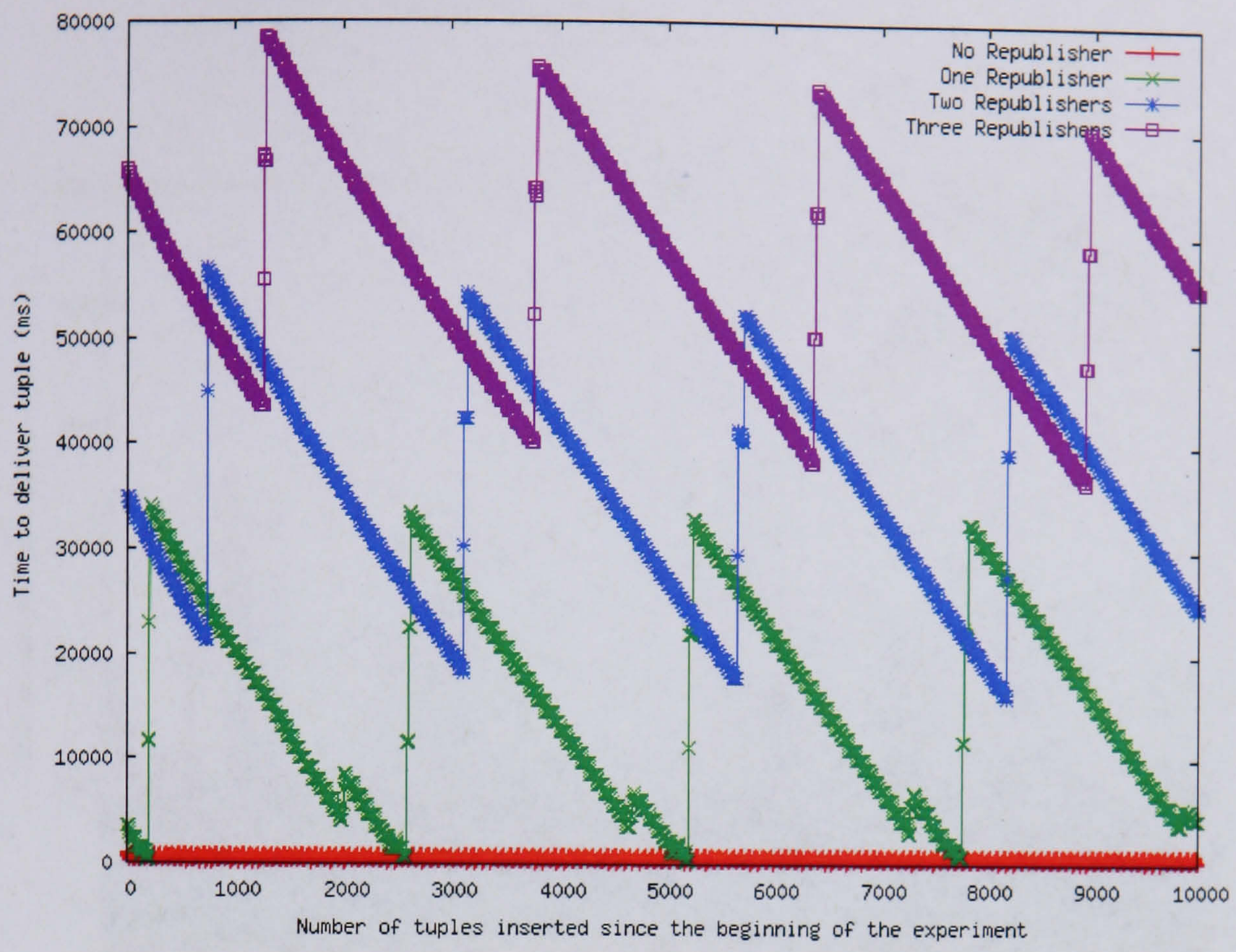


Figure 8.14: Time taken to deliver tuples from the producer to the consumer using different numbers of republishers when the producer introduces no delay between inserting each tuple.

new tuples without any delay between polls.

Figure 8.15 and Table 8.9 present the results when the republishers did not wait between each round of polling for tuples. The graph clearly shows that the introduction of each level of republisher increases the throughput time of a tuple. However, these graphs show more constant performance since the republishers are no longer introducing a delay.

The average delivery times show that one republisher introduces a delay of 23 ms, a second republisher adds 18 ms to the delay, and the third republisher adds 15 ms. The delay introduced by each republisher is the time taken for:

1. The tuple to be sent over the LAN to the republisher's consumer agent.

2. The republisher to retrieve the tuple from its consumer agent.

3. The republisher to publish the tuple through its producer agent.

The reduction in additional delay by each level of republisher is likely to be due to tuples getting grouped together and the synchronisation of the republishers polling

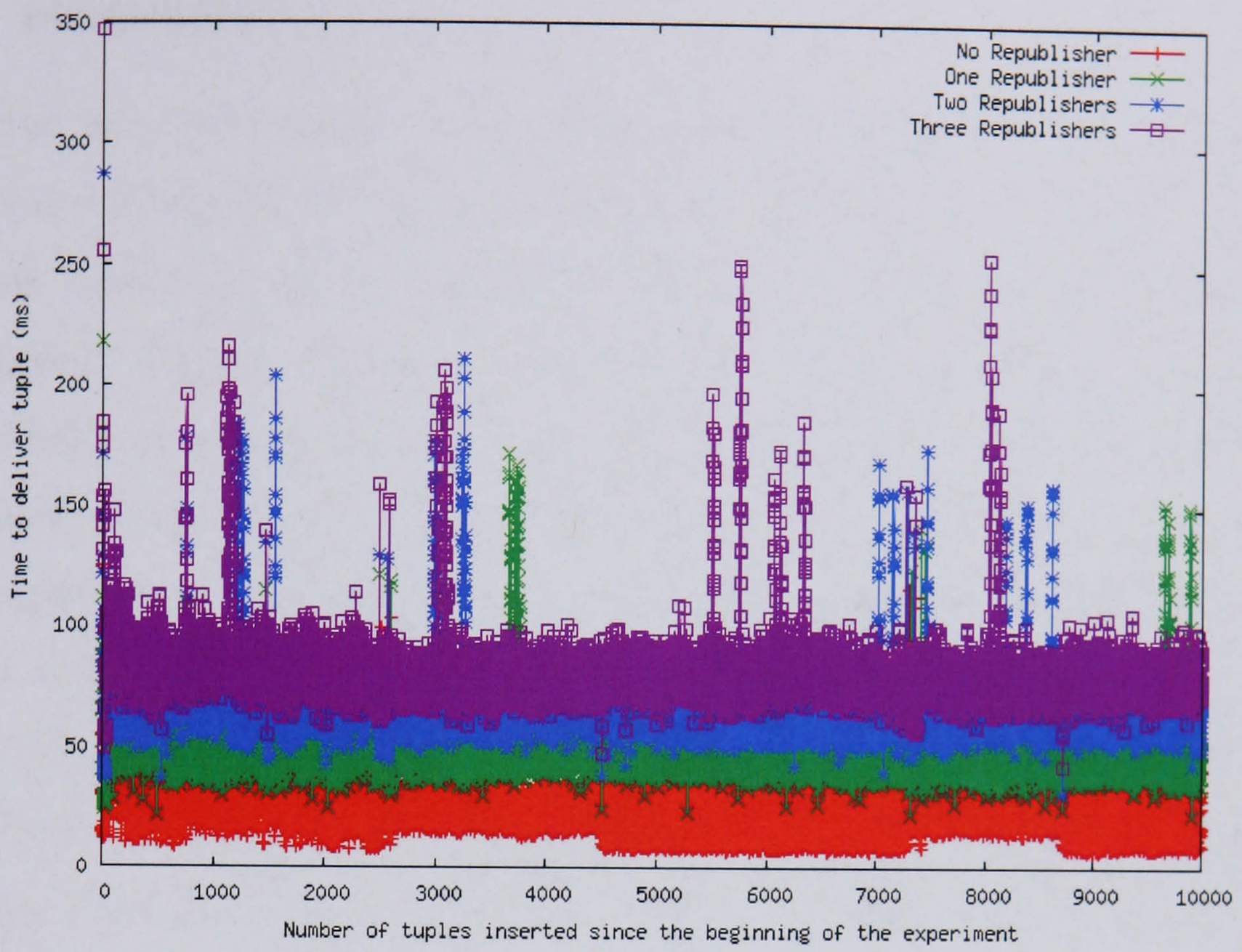


Figure 8.15: Time taken to deliver tuples from the producer to the consumer using different numbers of republishers which introduce no delay between polls of their input queues.

Number of Republishers	Average time to deliver a tuple (ms)	Variance
0	26	10
1	49	11
2	67	13
3	82	15

Table 8.9: Average time to deliver a tuple using different numbers of republishers; no delays are introduced by the republishers between each poll of their input queues.

their input queue.

8.2.4 Discussion

As expected, the results of the experiments showed that for each level of republishers a delay was introduced to the delivery time of a tuple. The extent of the delay introduced, within the experimental conditions used, was governed by the delay in the republisher between each successive poll of its consumer agent for tuples. When the delay between successive polls was eliminated, the time taken to deliver a tuple by each level of republishers became constant and was small, less than 25 ms.

The experiments were conducted on machines linked by a 100 Mbps network with the result that network delays were minimal. The purpose of creating a publisher hierarchy was to allow the R-GMA Grid information and monitoring system to scale to large Grids. It is anticipated that such Grids will incorporate machines located at multiple sites with wide area network links. In such a setting, it is likely that larger network delays would be introduced when sending tuples between sites. Thus, it would seem sensible to set up a republisher at each site to collect together the data from that site and make it available to others from a single location at that site.

It would have been desirable to investigate the benefits for query answering of using a republisher to gather the data from several producers. However, such an experiment would have been very complex and prone to experimental errors. While it has not been possible to show the benefits of the partial answers of the republishers experimentally, it is possible to show the significant reduction in the number of connections that each consumer would need to maintain. Note that there is a performance cost in maintaining a connection both on the side of the producer and on the side of the consumer.

Suppose there are m consumers all posing similar queries for which there are n relevant producers. When there are no relevant republishers available for the queries then each of the m consumers must contact each of the n producers. This means that the total number of connections is given by

$$n.m \tag{8.9}$$

Now suppose there are k non-overlapping relevant republishers that, for each of meta

query plans generated by the consumers form their own equivalence class. Each of the k republishers must maintain n/k connections while the consumers must each maintain k connections. Thus, the total number of connections is given by

$$\frac{k.n}{k} + k.m = n + k.m \tag{8.10}$$

Finally, when each of the equivalence classes in each of the query plans for the consumers contain l republishers, then each of the l republishers must maintain n/k connections but the number of connections that the consumers must maintain does not increase. Thus, the total number of connections is given by

$$\frac{l.k.n}{k} + k.m = l.n + k.m \tag{8.11}$$

To illustrate the gain from using the republishers consider that there are 100 consumers with similar queries for which there are 1,000 producers, then the total number of connections would be 100,000. When there are just 10 non-overlapping republishers the total number of connections reduces to 2,000, and when the redundancy of 5 copies of each of the republishers exist the number of connections increases to only 6,000. Obviously, to get the maximum benefit from the republishers the queries that they pose needs to be carefully chosen.

8.3 Summary

This chapter has looked at two aspects of the performance of the implementation of the stream integration system.

The first considered the performance of the registry service which needs to be very efficient so that it does not impact the performance of the whole system. The results showed that there are significant performance gains in using a structured representation for the conditions of a components query. The results of these experiments were used to guide the implementation of the query planning mechanisms.

The second set of performance measures looked at the delay introduced in a tuple reaching a consumer by using a hierarchy of republishers. Delays are introduced as the tuple must pass through a number of republishers before reaching the consumer. It was shown that the time taken by the republishers is very small. about 20 ms. although

if the current implementation of a republisher were able to be used for answering a continuous query it would introduce a greater delay with large variability due to the delay between successive polls of its consumer agent. It was also shown that there is a significant reduction in the number of connections that a consumer and a producer must maintain when using a hierarchy of publishers.

if the current implementation of a republisher were able to be used for answering a continuous query it would introduce a greater delay with large variability due to the delay between successive polls of its consumer agent. It was also shown that there is a significant reduction in the number of connections that a consumer and a producer must maintain when using a hierarchy of publishers.

Chapter 9

Conclusions

The previous chapters have provided an architecture for integrating distributed data from multiple autonomous sources, a formal framework for planning and maintaining the execution of a continuous query, and details of a prototype implementation of these techniques. This chapter will briefly summarise the main results of the work, and then suggest future work which will build upon these results.

9.1 Review of Thesis and Conclusions

The goal of this thesis was to develop mechanisms by which multiple autonomous distributed data sources, both streaming and stored, could be queried in a unified and efficient manner without the user needing to know specific details about the existence of individual sources or their locations. To achieve this goal a data integration approach was adopted, whereby a user poses a query over a global schema which is then translated into one or more queries over the relevant available data sources.

Previous work on data integration has only considered stored data sources, and the one-off queries associated with such data sources, i.e. the sources would present their data as if it existed in a database and the data would not be updated during the course of a query. However, the focus of this work has been on allowing streams of data to be included in a data integration setting and the continuous queries used to access such data. Thus, new techniques and mechanisms were needed to integrate data where some of the distributed data sources publish their data as a stream.

This thesis has proposed an architecture for publishing and querying both stream-

ing and stored data. As a first step to realising this architecture, mechanisms for answering continuous selection queries over the global schema have been developed. Since such queries are often long-lived, techniques for maintaining the answer streams when the configuration of the system change have also been developed.

The stream integration system architecture proposed in this thesis consisted of five types of components:

Schema Service: Maintains details of the relations that form the global schema. The relations were separated into stream and database relations.

Producer: Allows a data source to publish data according to a view description.

Consumer: Allows data to be retrieved that satisfies a query over the global schema.

Republisher: Poses a query over the global schema and publishes the answer for use in answering other queries.

Registry Service: Maintains details of the producers, consumers, and republishers that are registered in the system to facilitate query planning.

The architecture was shown to meet the requirements of the motivating application, a Grid information and monitoring system, although it is a general architecture and could be used for any integration application where both streaming and stored data needs to be accessible. In particular, the architecture is scalable since the data published in the system flows from the producers, through zero or more republishers, and then to the consumers, it does not flow through a centralised component.

A key feature of any data integration system is its query planning mechanism. The query planning mechanism is responsible for transforming the query over the global schema into a set of queries over the available data sources such that a sound, and in some cases complete, answer to the global query is returned. For a stream integration system, it was shown that the query planning mechanism should also ensure two further properties. The first of these is that the answer stream should guarantee some sort of *order* property. However, an exact chronological ordering is not possible since the streams originate from multiple distributed sources which will inevitable lead to some discrepancy in timestamps based on the drift of local clocks.

Instead, a *weak order* property, which ensures that the tuples with the same key values are in chronological order, was shown to be desirable. The second further property that a stream integration query planning mechanism should ensure is that the answer streams are duplicate free. Although it was assumed that the streams published by the producers are disjoint, the republishers introduce redundancy and thus make it possible for duplicates to appear in the answer stream.

A mechanism for generating query plans for answering continuous selection queries that favoured the use of republishers over producers was developed in this thesis. The query plans were derived from meta query plans which contain groups of maximal relevant republishers that can provide equivalent data for a query and those maximal relevant producers which provide additional data. It was shown that the approach generated query plans which are guaranteed to generate answer streams with the four desirable properties, i.e. the answer stream for a continuous query will be sound and complete with respect to the query, duplicate free, and weakly ordered. These query plans are computed by the consumer interacting with the registry service.

However, the meta query plans and query plans generated by the developed query answering mechanism do not consider the locality of the publishers. Thus, it is possible for a consumer and a relevant producer to be closely located but for the data to travel vast distances since the consumer uses a distant republisher. This of course depends on the deployment of the system. For the Grid information and monitoring application used to motivate this work, it was suggested in Chapter 8 that a suitable deployment of a republisher would be near several producers, e.g. a site with several producers would have a site level republisher. This republisher would be used to merge the streams together in the locality of their sources and make the merged stream available. A consumer in the same locality could then exploit the benefits of the republisher without the performance cost of sending the data over large distances.

Since continuous queries are long-lived, it is possible that during the execution of a query there can be a change in the set of available publishers, i.e. the producers and republishers. Such a change can affect the answer stream generated for a query. As such, a mechanism was developed to identify which query plans are potentially affected when there is a change, and to update the plans when they are affected. Due to the fact that query plans are based on meta query plans containing maximal

relevant publishers, for most updates either no change is required or only a minimal change of substituting one equivalent republisher for another. For the two cases where a substantial change to the query plan is required, these can be performed entirely by the information contained in the consumer.

A key feature of the proposed architecture is the republisher component. The role of the republishers is twofold. First, they facilitate complex one-time queries over several streams by archiving the data streams. In particular, for the case of history queries the republishers allow queries to be posed that span substantial periods of historic data that has been published by several producers.

Secondly, for continuous queries the republishers facilitate more efficient query answering as they merge the streams of several publishers into one stream, thus reducing the number of publishers that a query needs to access. This can result in a hierarchy of publishers being formed as it is possible for a republisher to consume its data from another republisher. If such a hierarchy is formed this will increase the time taken for a tuple to reach a consumer from the producer which publishes it as it must travel through more components. Experiments were run to investigate the length of this delay over the prototype implementation of the system deployed on several machines connected by a high speed local area network. The results showed that the delay introduced by a republisher was in the region of 20 ms when the republishers continuously poll their input queue.

9.2 Overcoming Incomplete Data

Another important aspect of the stream integration system is that of handling incompleteness in the data. Initial work on this topic has been conducted and reported in [115–117].

Incompleteness can occur for many reasons. Four categories for the sources of incompleteness when publishing data, in particular stream data, on a Grid have been identified [115]. These are:

Data Source Incompleteness: The sources of data do not contain all the data that they claim. This can be because of using inappropriate or out of date schemas resulting in the use of null values, or it can be because the data source does not

contain everything that it claims.

Data Integration Incompleteness: When data sources make their data available through some global schema incompleteness can occur due to the integration process. This can be because the sources are not able to accurately describe their content using the global schema due to limitations in the description language, or the query answering mechanisms limit the ability of the integration system to retrieve data.

Distribution of Data Sources: When data sources are distributed, e.g. publishing data on a Grid, communication and reliability issues can give rise to incompleteness. For example, components can fail leading to data not being available or communication errors can occur.

Incompleteness with respect to a Query: The data repositories on a Grid may not contain all the data that a query requests, e.g. a history query that requests data that is older than that stored by the republishers. Such a query cannot be answered completely although some form of partial answer may be generated by the republishers.

Work has been conducted to overcome the third category of incompleteness, specifically when the incompleteness manifests itself as missing values, when answering a history query [116, 117]. When a data stream which has missing values is archived by a republisher, the values will also be missing from the archive. Thus, the system requires techniques for:

1. Detecting when a data stream arriving at a republisher is missing values.

2. Storing details about the missing values.

3. Answering queries over data stores containing missing values.

It is possible to detect when a data stream arriving at a republisher is missing values on a specific channel. If the channel on the stream is periodic, i.e. it is published with a certain frequency, then a republisher can expect to receive tuples at certain time intervals. If a tuple does not arrive within a time period then it can be assumed to be missing. However, if the channel on the stream is irregular then a republisher

can still detect when tuples are missing by comparing the communication sequence number with the last received communication sequence number.

Both detection mechanisms allow a republisher to detect when one or more tuples are missing on a specific channel and the number of tuples that are missing. This means that the republisher knows some information about missing tuples, i.e. the values that define the channel and potentially the timestamp. This information can be represented by inserting a tuple containing null values for the measurement values of the tuple. For example, consider again the ntp relation, from Chapters 4, 5, and 6, with the schema

$$\mathsf{ntp}(\underline{\mathsf{from}}, \underline{\mathsf{to}}, \underline{\mathsf{tool}}, \underline{\mathsf{psize}}, \mathsf{latency}, [\mathsf{timestamp}]). \tag{9.1}$$

If a tuple is detected as missing using the irregular stream detection method on the Heriot-Watt to Rutherford Appleton Laboratory channel that used the PingER tool with packets of 256 bytes, then the following representative tuple could be inserted

$$(\texttt{'hw'}, \texttt{'ral'}, \texttt{'ping'}, 256, \texttt{null}, \texttt{null}). \tag{9.2}$$

Such representative tuples would then also be stored in the archive of the data stream.

When data stream archives can contain representative tuples, the query answering mechanisms must be extended to enable the information contained in a representative tuple to be exploited. The ideas of certain, possible, and impossible answer sets [118] were extended to permit representative tuples to be used to generate an answer to a query and possibly be returned as part of an answer to a query. This allowed some queries to be answered completely even though there was incomplete information in the relevant data. The mechanism also allows the user to be informed when it is not possible to answer a query completely, and to provide additional information about the missing parts of the answer.

9.3 Future Work

The work of this thesis gives rise to a number of interesting future research and development directions. Some of these will need to be followed before the work can be applied in real world applications, e.g. protocols to allow a consumer to switch

from one query plan to another. Others will increase the power of the system and thus make a stream integration system a more useful tool to a larger number of users, e.g. increasing the expressivity of the query language supported. Some of the issues that need to be addressed are discussed below.

9.3.1 Generating Query Plans

An important issue in the implementation of the stream integration system is the choice of query plan generated from the meta query plan. The implementation developed for this thesis simply chooses a republisher at random from an equivalence class containing more than one republisher. However, in a deployment there could be significant benefit from employing a cost model to choose between the republishers in an equivalence class. Such a cost model could be based on locality of the republisher and thus reducing communication times. Another cost model could be based on the monetary cost of using the service provided by a republisher if they charge for their data. Any such model would need to be developed for the application domain where the integration system was to be deployed.

It would also be interesting to investigate the effects of locality on the choice of relevant publishers for a query plan. The work of this thesis assumed that using republishers as high up the hierarchy as possible would be of maximum benefit since in the Grid monitoring application this would give stability against changes in the set of producers. The mechanism also ensured that the top level republishers do not become overloaded by identifying equivalent republishers for a specific query lower down in the hierarchy. However, if the chosen republisher is significantly further away from the consumer than the producers generating the streams then there are issues of locality that should be investigated.

9.3.2 Protocols for Plan Switching

Another important area of development work that would need to be conducted before deploying a real world version of the stream integration system would be to devise appropriate protocols and infrastructure to allow a consumer to switch seamlessly from one query plan to another. The prototype developed for this thesis has a mechanism

to update the query plan which consists of creating the new connections and then removing the old connections. During this process, it is possible that the answer stream for a query will contain duplicates (as it is consuming data from two overlapping sources) and/or not be complete (as the component that it was consuming from no longer exists). Section 7.4 proposed one approach that could be followed. However, the development of these protocols would depend on the application domain and deployment, and as such were beyond the scope of this thesis. The development of such protocols would allow the query planning and maintenance techniques to be adopted by the R-GMA Grid information and monitoring system.

9.3.3 Increased Query Functionality

The query planning element of this thesis has considered how to answer continuous selection queries. When these techniques are coupled with the one-time querying techniques of the R-GMA system [108], complex one-time queries involving joins between multiple streams and aggregation operators can be answered. However, the proposed architecture allows both streaming and stored data sources to publish data. To fully realise this functionality, additional query planning techniques are required to permit continuous queries involving joins [119–121], aggregation [68, 122], and windowing operators [67–69].

In order that joins between arbitrary data can be processed there needs to be a formal understanding of the semantics of a join involving a stream. This needs to be understood both for a join involving two streams, and for a join between a stream relation and a stored relation. For example, does a join between two streams, s_1 and s_2, mean that every tuple in s_1 should be considered as a candidate for a join with a tuple in s_2? Bear in mind that data streams are potentially infinite and so such a join could never be computed. As such, a join involving a stream needs to include some sort of window to declare the extent of data that is considered for the join. To be able to declare such windows requires additional constructs in the query language and would need to be considered when planning the execution of a query, i.e. for a publisher to be used to answer a query involving a window it must have a suitable window of data. Work has already been conducted on executing joins over windows [119–121]. However, these would need to be extended for an integration

setting.

Once the semantics of a join have been formalised there is the issue of where to compute such joins. In order to decide where to compute such operations the query planner must be able to reason about the query processing capabilities of the publishers. For example, a producer that publishes stored data from a database system that has been enhanced to process a continuous join with a stream would need to declare that it is capable of such an operation so that the query planning does not generate a plan that ships the stored data out of the publisher. When a publisher of stored data is not capable of processing such joins then mechanisms for republishing the data would be required. This results in issues in ensuring that the republished copy is kept up-to-date *cf. view maintenance* [123–125], i.e. if the data in the original data source changes then the republished data should also be updated, but this mechanism should not become a burden on the system itself as this would affect the overall performance of the system.

Another area which will require further research, both on the semantics and the execution, is that of computing aggregate operators over a stream. The idea of an aggregate query over a database is to condense the data so that the query returns a smaller amount of information. However, the approaches involving data streams published in the literature for performing aggregation functions, such as computing the average over a sliding window, will still result in the same number of tuples in the answer stream as in the original data stream. This is because every time a new tuple arrives and the window slides along the stream by one tuple, then the average must be recalculated and the answer streamed back. Thus, as each new tuple arrives on the stream a new answer tuple is generated. Whilst techniques for storing approximate values for these aggregates in limited memory have been developed [126], there has been little work on condensing the aggregate streams. A notable exception is [127]. However, once these techniques have been developed, they will need to be combined with the work on aggregate queries in a data integration setting [128, 129].

9.3.4 Technology Uptake

At present users of the R-GMA system only make limited use of the continuous query features and as such there is not the demand for the performance gains in using a

hierarchy of publishers. For the publisher hierarchy to be of benefit, there would need to be a large number of consumers posing continuous queries.

The limited use of the continuous query features in the R-GMA system is probably due to two reasons. The first is a lack of experience and knowledge of stream processing. Research on the data stream paradigm only started in the mid to late 1990s. While there has been a lot of research interest in the area, currently there is only one commercial stream processing system on the market, StreamBase[1], which was first released in 2004, and limited understanding of how to query a stream. Secondly, the limitation of only being able to pose selection queries over a data stream limits the type of applications that can benefit, e.g. a visualisation tool needing to be updated with the progress of a job can currently benefit [130].

By increasing the querying functionality of the stream integration system to incorporate stream and stored data and to allow aggregate operations over that data, a broad range of new application areas become available. These include monitoring patients in their own environment [131], environmental monitoring [5,6], and managing disaster situations [132]. In the near future, there will be a plethora of disparate sensing devices which are available in a pervasive manner. A system that is capable of gathering these streams, inter-relating the data that they contain, and allowing the data to be combined with data in stored sources will become very important. With the future work outlined above building upon the results of this thesis, the proposed stream integration system would be capable of fulfilling this role.

[1] `http://www.streambase.com` (March 2007)

Bibliography

[1] J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. *SIGMOD Record*, 29(2):379–390, June 2000.

[2] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 28)*, pages 358–369, Hong Kong (China), August 2002. Morgan Kaufmann Publishers.

[3] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. In *IEEE Communications Magazine*, pages 102–114. IEEE Computer Society, August 2002.

[4] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of 2nd International Conference on Mobile Data Management*, volume 1987 of *Lecture Notes in Computer Science*, pages 3–14, Hong Kong (China), January 2001. Springer-Verlag.

[5] D. Hughes, P. Greenwood, G. Blair, G. Coulson, F. Pappenberger, P. Smith, and K. Beven. An intelligent and adaptable grid-based flood monitoring and warning system. In *Proceedings of the UK e-Science All Hands Meeting 2006*, Nottingham (UK), September 2006.

[6] *Proceedings of 4th Global Biodiversity Information Facility Science Symposium 2006*, Cape Town (South Africa), April 2006.

[7] A. Arasu, M. Cherniack, E.F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management

benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 30)*, pages 480–491, Toronto (Canada), August 2004. Morgan Kaufmann Publishers.

[8] S. Babu, L. Subramanian, and J. Widom. A data stream management system for network traffic managment. In *Proceedings of Workshop on Network-Related Data Management (NRDM 2001)*, May 2001.

[9] M. Swany and R. Wolski. Representing dynamic performance information in grid environments with the network weather service. In *Proceedings of 2nd International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 48–56, Berlin (Germany), May 2002. IEEE Computer Society.

[10] A.J. Wilson, R. Byrom, L.A. Cornwall, M.S. Craig, A. Djaoui, S.M. Fisher, S. Hicks, R.P. Middleton, J.A. Walk, A. Cooke, A.J.G. Gray, W. Nutt, J. Magowan, J. Leake, R. Cordenonsi, N. Podhorszki, B. Coghlan, S. Kenny, O. Lyttleton, and D. O'Callaghan. Information and monitoring services within a grid environment. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP)*, Interlaken (Switzerland), September 2004.

[11] P.J. Stockreisser, J. Shao, W.A. Gray, and N.J. Fiddian. Supporting QoS monitoring in virtual organisations. In *Proceedings of 4th International Conference on Service Oriented Computing (ICSOC 2006)*, volume 4294 of *Lecture Notes in Computer Science*, pages 447–452, Chicago (IL, USA), December 2006. Springer-Verlag.

[12] I.T. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 2nd edition, 2004.

[13] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[14] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report 574, Indiana University, Department of Computer Science, Indiana University. Bloomington, Indiana, USA., May 2003.

[15] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, pages 1–16, Madisson (WI, USA), June 2002. ACM Press. Extended version available from authors.

[16] L. Golab and M.T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.

[17] M. Stonebraker, U. Çetintemel, and S.B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, December 2005.

[18] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proceedings of 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, pages 277–289, Asilomar (CA, USA), January 2005. On-line proceedings.

[19] B. Plale and K. Schwan. Dynamic querying of streaming data with the dQUOB system. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):422–432, April 2003.

[20] B. Stegmaier, R. Kuntschke, and A. Kemper. StreamGlobe: Adaptive query processing and optimization in streaming P2P environments. In *Proceedings of the 1st International Workshop on Data Management for Sensor Networks (DMSN 2004)*, pages 88–97, Toronto (Canada), August 2004.

[21] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, 2nd edition, 1999.

[22] R. Yahyapour. Grid scheduling use cases. Informational GFD.I064, Global Grid Forum, http://www.ggf.org, March 2006.

[23] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[24] J.D. Ullman. Information integration using logical views. In *Proceedings of 6th International Conference on Database Theory (ICDT 1997)*, volume 1186

of *Lecture Notes in Computer Science*, pages 19–40, Delphi (Greece), January 1997. Springer-Verlag.

[25] A.Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, December 2001.

[26] M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, pages 233–246, Madisson (WI, USA), June 2002. ACM Press.

[27] D.B. Terry, D. Goldberg, D. Nichols, and B.M. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 321–330, San Diego (CA, USA), June 1992. ACM Press.

[28] A. Halevy, A. Rajaraman, and J. Ordile. Data integration: The teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 32)*, pages 9–16, Seoul (Korea), September 2006. ACM Press.

[29] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

[30] A.Y. Levy. Logic-based techniques in data integration. In *Logic-based Artificial Intelligence*, section Applications of Theorem Proving and Logic Programming, pages 575–595. Kluwer Academic Publishers, Norwell (MA, USA), 2000.

[31] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 2nd edition, 1982.

[32] A.Y. Levy, A. Rajaraman, and J.J. Ordille. Query-answering algorithms for information agents. In *Proceedings of 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference*, volume 1, pages 40–47, Portland (OR, USA), August 1996. MIT Press.

[33] A.Y. Levy, A. Rajaraman, and J.J. Ordille. Querying hetergeneous information sources using source descriptions. In *Proceedings of the 22nd International*

Conference on Very Large Data Bases (VLDB 22), pages 251–262, Mumbai (Bombay, India), September 1996. Morgan Kaufmann Publishers.

[34] X. Qian. Query folding. In *Proceedings of 12th International Conference on Data Engineering (ICDE 1996)*, pages 48–55, New Orleans (LA, USA), February 1996. IEEE Computer Society.

[35] P. Mitra. An algorithm for answering queries efficiently using views. In *Proceedings of 12th Australasian Database Conference (ADC2001)*, pages 99–106, Queensland (Australia), January 2001. ACM Press.

[36] O.M. Duschka, M.R. Genesereth, and A.Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, April 2000.

[37] R. Pottinger and A.Y. Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198, September 2001.

[38] E. Schallehn, K.-U. Sattler, and G. Saake. Efficient similarity-based operations for data integration. *Data and Knowledge Engineering*, 48(3):361–387, March 2004.

[39] H.T. El-Khatib, M.H. Williams, L.M. MacKinnon, and D.H. Marwick. A framework and test-suite for assessing approaches to resolving heterogeneity in distributed databases. *Information and Software Technology*, 42(7):505–515, May 2000.

[40] H.T. El-Khatib, M.H. Williams, D.H. Marwick, and L.M. MacKinnon. Using a distributed approach to retrieve and integrate information from heterogeneous distributed databases. *The Computer Journal*, 45(4):381–394, 2002.

[41] A.P. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys.* 22(3):183–236, September 1990.

[42] W.H. Inmon and R.D. Hackathorn. *Using the Data Warehouse.* John Wiley and Sons, 1994.

[43] R. Fagin, P.G. Kolaitis, R.J. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theorectical Computer Science*, 336(1):89–124, May 2005.

[44] P.G. Kolaitis. Schema mappings, data exchange and metadata management. In *Proceedings of 24th ACM Symposium on Principles of Database Systems (PODS 2002)*, pages 61–75, Balitmore (MD, USA), June 2005. ACM Press.

[45] A. Doan, N.F. Noy, and A.Y. Halevy (editors). Special section on semantic integration. *SIGMOD Record*, 33(4):11–70, December 2004.

[46] M.R. Stonebraker, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[47] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Loire, and T.G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston (MA, USA), May 1979. ACM Press.

[48] J.M. Hellerstein, M.J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M.A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[49] M. Antonioletti, M.P. Atkinson, R. Baxter, A. Borley, N.P.C. Hong, B. Collins, N. Hardman, A.C. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N.W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead. The design and implementation of Grid database services in OGSA-DAI. *Concurrency and Computation – Practice and Experience*, 17(2-4):357–376, February 2005.

[50] J. Smith, A. Gounaris, P. Watson, N.W. Paton, A.A.A. Fernandes, and R. Sakellariou. Distributed query processing on the grid. *International Journal of High Performance Computer Applications*, 17(4):353–368, November 2003. Extended version of the paper that appears in the proceedings of Grid Computing 2002.

[51] M.R. Genesereth, A.M. Keller, and O.M. Duschka. Infomaster: An information integration system. In *Proceedings of the 1997 ACM SIGMOD International*

Conference on Management of Data, pages 539–542, Tuscan (AZ, USA), June 1997. ACM Press.

[52] N. Leone, G. Greco, G. Ianni, V. Lio, G. Terracina, T. Eiter, W. Faber, M. Fink, G. Gottlob, R. Rosati, D. Lembo, M. Lenzerini, M. Ruzzi, E. Kalka, B. Nowicki, and W. Staniszkis. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 915–917, Baltimore (MD, USA), June 2005. ACM Press.

[53] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J.D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Date models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, March/April 1997.

[54] M.K Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of 18th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 53–61, Atlanta (GA, USA), May 1999. ACM Press.

[55] C. Raiciu and D.S. Rosenblum. Enabling confidentiality in content-based publish/subscribe infrarstructures. Technical Report RN/05/30, University College London, Department of Computer Science, University College London, London, 2005.

[56] C. Raiciu, D.S. Rosenblum, and M. Handley. Revisiting content-based publish/subscribe. In *Proceedings of 26th International Conference on Distributed Computing Systems Workshops (ICDCS 2006 Workshops)*, page 19, Lisboa (Portugal), July 2006. IEEE Computer Society.

[57] G. Eisenhauer, F.E. Bustamante, and K. Schwan. Event services in high performance systems. *Cluster Computing*, 4(3):243–252, July 2001.

[58] Y. Zhao, D.C. Sturman, and S. Bhola. Subscription propagation in highly-available publish/subscribe middleware. In *Proceedings of International Middle-*

ware Conference (Middleware 2004), volume 3231 of *Lecture Notes in Computer Science*, pages 274–293, Toronto (Canada), October 2004. Springer-Verlag.

[59] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.

[60] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, October 2002.

[61] A.I.T. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350, Heidelberg (Germany), November 2001. Springer-Verlag.

[62] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[63] D. Wu, Y.T. Hou, W. Zhu, Y.-Q. Zhang, and J.M. Peha. Streaming video over the internet: approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):282–300, March 2001.

[64] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S.B. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[65] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The stanford data stream management system. In *Data-Stream Management: Processing High-Speed Data Streams*. Springer-Verlag, New York (NY, USA), 2007. To appear.

[66] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S.R. Madden, V. Raman, F. Reiss, and M.A. Shah.

TelegraphCQ: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, March 2003.

[67] A.P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of 13th International Conference on Data Engineering (ICDE 1997)*, Birmingham (UK), April 1997. IEEE Computer Society.

[68] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. *SIGMOD Record*, 30(2):13–24, June 2001.

[69] L. Ma and W. Nutt. Frequency operators for condensative queries over data streams. In *Proceedings of IEEE International Conference on e-Business Engineering 2005 (ICEBE)*, pages 518–526, Beijing (China), October 2005. IEEE Computer Society.

[70] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[71] Characteristics of a good language for stream processing. White paper from `http://streamsql.org/files/Documentation/streamprocessing_language.pdf`.

[72] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E.F. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S.B. Zdonik. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, December 2004.

[73] R. Avnur and J.M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, Dallas (TX, USA), May 2000. ACM Press.

[74] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, and M.J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of 19th International Conference on Data Engineering (ICDE 2003)*, pages 25–36, Bangalore (India), March 2003. IEEE Computer Society.

[75] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, March 2002.

[76] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar (CA, USA), January 2003. On-line proceedings.

[77] I.T. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computer Applications*, 15(3):200–222, August 2001.

[78] I.T. Foster and C. Kesselman. Computational grids. In *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2, pages 15–51. Morgan Kaufmann Publishers, San Francisco (CA, USA), 1999.

[79] J. Gomes, M. David, J. Martins, L. Bernardo, A. García, M. Hardt, H. Kornmayer, J. Marco, D. Rodríguez, I. Diaz, D. Cano, J. Salt, S. Gonzalez, J. Sánchez, F. Fassi, V. Lara, P. Nyczyk, P. Lason, A. Ozieblo, P. Wolniewicz, M. Bluj, K. Nawrocki, A. Padee, W. Wislicki, C. Fernández, J. Fontán, Y. Cotronis, E. Floros, G. Tsouloupas, W. Xing, M.D. Dikaiakos, J. Astalos, B.A. Coghlan, E. Heymann, M.A. Senar, C. Kanellopoulos, A. Ramos, and D. Groen. Experience with the international testbed in the CrossGrid project. In *Proceedings of Advances in Grid Computing, European Grid Conference (EGC 2005)*, volume 3470 of *Lecture Notes in Computer Science*, pages 98–110, Amsterdam (The Netherlands), February 2005. Springer-Verlag.

[80] E. Laure (editor). The EU DataGrid setting the basis for production grids. *Journal of Grid Computing*, 2(4), April 2004.

[81] F. Gagliardi, B. Jones, F. Grey, M.-E. Bégin, and M. Heikkurinen. Building an infrastructure for scientific grid computing: status and goals of the EGEE project. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 363(1833):1729–1742. August 2005.

[82] I.T. Foster, J. Gieraltowski, S. Gose, N. Maltsev, E.N. May, A. Rodriguez, D. Sulakhe, A. Vaniachine, J. Shank, S. Youssef, D. Adams, R. Baker, W. Deng, J. Smith, D. Yu, I. Legrand, S. Singh, C. Steenberg, Y. Xia, M.A. Afaq, E. Berman, J. Annis, L.A.T. Bauerdick, M. Ernst, I. Fisk, L. Giacchetti, G.E. Graham, A. Heavey, J. Kaiser, N. Kurpopatkin, R. Pordes, V. Sekhri, J. Weigand, Y. Wu, K. Baker, L. Sorrillo, J. Huth, M. Allen, L. Grundhoefer, J. Hicks, F. Luehring, S. Peck, R. Quick, S. Simms, G. Fekete, J. vandenBerg, K. Cho, K. Kwon, D. Son, H. Park, S. Canon, K. Jackson, D.E. Konerding, J. Lee, D. Olson, I. Sakrejda, B. Tierney, M. Green, R. Miller, J. Letts, T. Martin, D. Bury, C. Dumitrescu, D. Engh, R. Gardner, M. Mambelli, Y. Smirnov, J.-S. Vöckler, M. Wilde, Y. Zhao, X. Zhao, P. Avery, R. Cavanaugh, B. Kim, C. Prescott, J.L. Rodriguez, A. Zahn, S. McKee, C.T. Jordan, J.E. Prewett, T.L. Thomas, H. Severini, B. Clifford, E. Deelman, L. Flon, C. Kesselman, G. Mehta, N. Olomu, K. Vahi, K. De, P. McGuigan, M. Sosebee, D. Bradley, P. Couvares, A. DeSmet, C. Kireyev, E. Paulson, A. Roy, S. Koranda, B. Moe, B. Brown, and P. Sheldon. The Grid2003 production Grid: Principles and practice. In *Proceedings of 13th International Symposium on High-Performance Distributed Computing (HPDC-13 2004)*, pages 236–245, Honolulu (HI, USA), June 2004. IEEE Computer Society.

[83] F. Berman. Viewpoint: From TeraGrid to knowledge grid. *Communications of the ACM*, 44(11):27–28, November 2001.

[84] I.T. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of Network and Parallel Computing, IFIP International Conference (NPC 2005)*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13, Beijing (China), December 2005. Springer-Verlag.

[85] B. Koblitz, T. Chen, W. Ueng, J. Herrala, M. Lamanna, D. Liko, A. Maier, J. Mockicki, A. Peters, F. Orellana, V. Pose, A. Demichev, and D. Feichtinger. Experiences with the gLite Grid middleware. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP)*, page 43, Interlaken (Switzerland), September 2004. Online proceedings. Paper ID 305.

[86] M.L. Massie, B.N. Chun, and D.E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[87] J. Basney and M. Livny. Managing network resources in condor. In *Proceedings of 9th International Symposium on High-Performance Distributed Computing (HPDC-9 2000)*, pages 298–299, Pittsburgh (PA, USA), August 2000. IEEE Computer Society.

[88] W. Matthews and L. Cottrell. The PingER project: Active internet performance monitoring for the HENP communitiy. *IEEE Communications Magazine*, 38(5):130–136, May 2000.

[89] A. Cooke, A.J.G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A. Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S. Fisher, N. Podhorszki, B. Coghlan, S. Kenny, and D. O'Callaghan. R-GMA: An information integration system for grid monitoring. In *Proceedings of On the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE—OTM Confederated International Conferences (OTM 2003)*, volume 2888 of *Lecture Notes in Computer Science*, pages 462–481, Catania (Italy), November 2003. Springer-Verlag.

[90] X. Zhang, J.L. Freschl, and J.M. Schopf. Scalability analysis of three monitoring and information systems: MDS 2, R-GMA, and Hawkeye. *IEEE Transactions on Parallel and Distributed Systems*. To appear.

[91] B. Plale, P. Dinda, and G. von Laszewski. Key concepts and services of a grid information service. In *Proceedings of 15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, Louisville (KY, USA), September 2002. International Society for Computers and their Applications.

[92] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A Grid monitoring architecture. Informational GFD.7, Global Grid Forum. http://www.ggf.org, January 2002.

[93] X. Zhang, J.L. Freschl, and J.M. Schopf. A performance study of monitoring and information services for distributed systems. In *Proceedings of 12th International Symposium on High-Performance Distributed Computing (HPDC-12 2003)*, pages 270–282, Seattle (WA, USA), June 2003. IEEE Computer Society.

[94] W. Smith. A system for monitoring and management of computational grids. In *Proceedings of 31st International Conference on Parallel Processing (ICPP 2002)*, pages 55–64, Vancouver (Canada), August 2002. IEEE Computer Society.

[95] W.E. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of NASA's information power grid. In *Proceedings of 8th International Symposium on High-Performance Distributed Computing (HPDC-8 1999)*, pages 197–204, Redondo Beach (CA, USA), August 1999. IEEE Computer Society.

[96] D. Gunter. An overview of the PyGMA. `http://sourceforge.net/docman/display_doc.php?docid=8662&group_id=28724`, 2004.

[97] R.L. Ribler, H. Simitci, and D.A. Reed. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, September 2001.

[98] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan. New grid scheduling and rescheduling methods in the GrADS project. *International Journal of Parallel Programming*, 33(2-3):209–229, June 2005.

[99] B. Balis, M. Bubak, W. Funika, T. Szepieniec, R. Wismüller, and M. Radecki. Monitoring grid applications with grid-enabled OMIS monitor. In *Proceedings of 1st European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 230–239, Santiago de Compostela (Spain), February 2003. Springer-Verlag.

[100] R. Wolski. Experiences with predicting resource performance on-line in computational grid settings. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):41–49, March 2003.

[101] H.-L. Truong and T. Fahringer. SCALEA-G: A unified monitoring and performance analysis system for the grid. *Scientific Programming*, 12(4):225–237, 2004.

[102] G. Gombás, C.A. Marosi, and Z. Balaton. Grid application monitoring and debugging using the Mercury monitoring system. In *Proceedings of Advances in Grid Computing, European Grid Conference (EGC 2005)*, volume 3470 of *Lecture Notes in Computer Science*, pages 193–199, Amsterdam (The Netherlands), February 2005. Springer-Verlag.

[103] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I.T. Foster. Grid information services for distributed resource sharing. In *Proceedings of 10th International Symposium on High-Performance Distributed Computing (HPDC-10 2001)*, pages 181–194, San Francisco (CA, USA), August 2001. IEEE Computer Society.

[104] J.M. Schopf, I. Raicu, L. Pearlman, N. Miller, C. Kesselman, I. Foster, and M. D'Arcy. Monitoring and discovery in a web services framework: Functionality and performance of globus toolkit MDS 4. In *Proceedings of 15th International Symposium on High-Performance Distributed Computing (HPDC-15 2006)*, Paris (France), June 2006. IEEE Computer Society.

[105] T.A. Howes, M.C. Smith, and G.S. Good. *Understanding and Deploying LDAP Directory Services*. Addison-Wesley, second edition, 2003.

[106] M. Gerndt, R. Wismüller, Z. Balaton, G. Gombás, P. Kacsuk, Z. Németh, N. Pordhorszki, H. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. Performance tools for the Grid: State of the art and future. Technical Report Vol. 30, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM), Technische Universitaet Muenchen, Shaker Verlag, 2004.

[107] I. Foster, K. Czajkowski, D.E. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and managing state in distributed systems: The role of OGSI and WSRF. *Proceedings of the IEEE*, 93(3):604–612, March 2005.

[108] A.W. Cooke, A.J.G. Gray, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, L. Field, S.M. Fisher, S. Hicks, J. Leake, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny, D. O'Callaghan, and J. Ryan. The relational grid monitoring architecture: Mediating information about the grid. *Journal of Grid Computing*, 2(4):323–339, December 2004.

[109] A. Cooke, A.J.G. Gray, and W. Nutt. Stream integration techniques for grid monitoring. *Journal on Data Semantics*, 2:136–175, 2005.

[110] S. Andreozzi, S. Burke, L. Field, S. Fisher, B. Kónya, M. Mambelli, J.M. Schopf, M. Viljoen, and A. Wilson. GLUE schema specification. http://infnforge.cnaf.infn.it/glueinfomodel/index.php/Spec/V12, December 2005.

[111] R. Byrom, B.A. Coghlan, A.W. Cooke, R. Cordenonsi, L. Cornwall, M. Craig, A. Djaoui, A. Duncan, S. Fisher, A.J.G. Gray, S. Hicks, S. Kenny, J. Leake, O. Lyttleton, J. Magowan, R. Middleton, W. Nutt, D. O'Callaghan, N. Podhorszki, P. Taylor, J. Walk, and A.J. Wilson. Fault tolerance in the R-GMA information and monitoring system. In *Proceedings of Advances in Grid Computing, European Grid Conference (EGC 2005)*, volume 3470 of *Lecture Notes in Computer Science*, pages 751–760, Amsterdam (The Netherlands), February 2005. Springer-Verlag.

[112] A.J.G. Gray and W. Nutt. Republishers in a publish/subscribe architecture for data streams. In *Proceedings of 22nd British National Conference on Databases (BNCOD22)*, volume 3567 of *Lecture Notes in Computer Science*, pages 179–184, Sunderland (UK), July 2005. Springer-Verlag.

[113] A.J.G. Gray and W. Nutt. A data stream publish/subscribe architecture with self-adapting queries. In *Proceedings of On the Move to Meaningful Internet*

Systems 2005: CoopIS, DOA, and ODBASE—OTM Confederated International Conferences (OTM 2005), volume 3760 of *Lecture Notes in Computer Science*, pages 420–438, Agia Napa (Cyprus), October 2005. Springer-Verlag.

[114] R. Byrom, B. Coghlan, A. Cooke, R. Cordenonsi, L. Cornwall, M. Craig, A. Djaoui, S. Fisher, A. Gray, S. Hicks, S. Kenny, J. Leake, O. Lyttleton, J. Magowan, R. Middleton, W. Nutt, D. O'Callaghan, N. Podhorszki, P. Taylor, J. Walk, and A. Wilson. Production services for information and monitoring in the grid. In *Proceedings of the UK e-Science All Hands Meeting 2004*, Nottingham (UK), September 2004.

[115] A.J.G. Gray, W. Nutt, and M.H. Williams. Sources of incompleteness in grid publishing. In *Proceedings of 23rd British National Conference on Databases (BNCOD23)*, volume 4042 of *Lecture Notes in Computer Science*, pages 94–101, Belfast (UK), July 2006. Springer-Verlag.

[116] A.J.G. Gray, M.H. Williams, and W. Nutt. Answering arbitrary conjunctive queries over incomplete data stream histories. In *Proceedings of 8th International Conference on Information Integration and Web-based applications and Services (iiWAS2006)*, pages 259–268, Yogyakarta (Indonesia), December 2006. Austrian Computer Society.

[117] A.J.G. Gray, W. Nutt, and M.H. Williams. Answering queries over incomplete data stream histories. *International Journal of Web Information Systems*, 2007. Invited submission that has been accepted to appear subject to minor revisions.

[118] G. Grahne and V. Kiricenko. Partial answers in information integration systesm. In *International Workshop on Web Information and Data Management (WIDM 2003)*, pages 98–101, New Orleans (LA, USA), November 2003. ACM Press.

[119] U. Srivastava and J. Widom. Memory-limited execution of window stream joins. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 30)*, pages 324–335, Toronto (Canada), August 2004. Morgan Kaufmann Publishers.

[120] B. Gedik, K.-L. Wu, P.S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM 2005)*, pages 171–178, Bremen (Germany), October 2005. ACM Press.

[121] X. Gu, P.S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *Proceedings of 23rd International Conference on Data Engineering (ICDE 2007)*, Istanbul (Turkey), April 2007. IEEE Computer Society.

[122] J. Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 311–322, Baltimore (MD, USA), June 2005. ACM Press.

[123] A. Gupta and I.S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, 1999.

[124] Y. Wei, S.H. Son, and J.A. Stankovic. Maintaining data freshness in distributed real-time databases. In *Proceedings of 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, pages 251–260, Catania (Italy), June 2004. IEEE Computer Society.

[125] N. Kim and S. Moon. Concurrent view maintenance scheme for soft real-time data warehouse systems. *Journal of Information Science and Engineering*, 23(3):725–741, May 2007.

[126] M. Datar and R. Motwani. The sliding-window computation model and results. In *Data Streams: Models and Algorithms*, chapter 8, pages 149–168. Springer-Verlag, 2007.

[127] L. Ma, W. Nutt, and H. Taylor. Condensative stream query language for data streams. In *Proceedings of 18th Australasian Database Conference (ADC07)*, Ballarat (Australia), January 2007. Australian Computer Society Press.

[128] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Transactions on Database Systems*, 31(2):672–715. June 2006.

[129] S. Cohen, W. Nutt, and Y. Sagiv. Deciding equivalences among conjunctive aggregate queries. *Journal of the ACM*, 54(2), April 2007.

[130] H. Thomson. 3D grid monitor debuts at UK e-Science meeting. Science Grid This Week On-line Magazine, September 2006. `http://www.interactions.org/sgtw/2006/0927/rtm_more.html`.

[131] M. Subramanian, A.S. Ali, O. Rana, A. Hardisty, and E.C. Conley. Healthcare@Home: Research model for patient-centred healthcare services. In *Proceedings of John Vincent Atanasoff International Symposium on Modern Computing*, pages 107–113, Sofia (Bulgaria), October 2006. IEEE Computer Society.

[132] D. Berry, A. Usmani, J.L. Torero, A. Tate, S. McLaughlin, S. Potter, A. Trew, R. Baxter, M. Bull, and M. Atkinson. FireGrid: Integrated emergency response and fire safety engineering for the future built environment. In *Proceedings of the UK e-Science All Hands Meeting 2005*, Nottingham (UK), September 2005.